



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

UNIVERSITY OF ALBERTA

Programming Using Constructive Proofs

BY

Andrew Walenstein



A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1992



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77296-4

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Andrew Walenstein

TITLE OF THESIS: Programming Using Constructive Proofs

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

... *Andrew Walenstein* ...

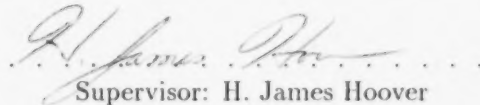
Permanent Address:
8216 186 Street
Edmonton, AB
T5T 1H4
Canada

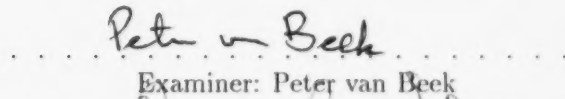
Date: *July 03, 1992*

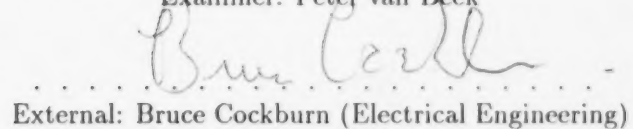
UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Programming Using Constructive Proofs** submitted by **Andrew Walenstein** in partial fulfillment of the requirements for the degree of Masters of Science.


Supervisor: H. James Hoover


Examiner: Peter van Beek


External: Bruce Cockburn (Electrical Engineering)

Date: July 3, 1992.

To Lorena

Abstract

Constructive logics and type theories can be viewed as programming environments when they are able to construct correct programs from proofs of their specifications. In this thesis we discuss the implementation of a system called MIZAR-C which allows one to write proofs in a classical first-order natural deduction logic from which it can extract programs when the proofs are constructive.

A major task involved in using a logic as a programming language is the encoding of the relationship between the specification and a program which meets it. We first lay down the foundation for a general framework for encoding program-specification relationships in MIZAR-C which follows from the definition of the mechanism to extract programs from proofs. The programs extracted from proofs are defined by a realizability interpretation for the inference rules in the logic. We will also need to introduce a marking scheme for the inference rules in order to show that allowing classical inference rules does not cause erroneous programs to be extracted: the marking scheme allows us to recognize non-constructive proofs. We also discuss the soundness of the extended features and conveniences of MIZAR-C's logic such as the equality inference rules, multi-way disjunction, and function terms.

Second, we describe how the realizability interpretation's definition outlines a kind of framework for extending the logic with axioms representing definitions of the basic data types which might be found in programming languages. That is, the restrictions which the realizability interpretation imposes on the realizations paired with axioms indicates how to add constructions such as data types and induction schemes to the system. We also introduce the possibility of modifying the realizability interpretation in order to expand this framework so that it may encode specifications for programs which execute non-deterministically and which may not behave purely functionally.

Finally, we conclude with a demonstration proof of the summation procedure for functions over natural numbers, and we mention problems which need to be resolved in order to provide a logical framework convenient enough for writing programs in general. The proof of the summation procedure serves to illustrate a need for abstraction methods such as tactics and abbreviations.

Acknowledgements

I would like to thank my supervisor, Jim Hoover, for his invaluable and inspiring tutelage, winning attitude, and constructive comments. He has provided a marvelous research environment in which the joys of comprehending new ideas were spiced with deliciously serendipitous discoveries. I thank him for adding depth to my academic pursuits by being a good friend, and often dragging me away from the terminal and off to movies and lunches. I believe he somehow managed to temper my often wild and impulsive ideas (and tolerated caffeine-induced, late-night e-rantings), and provided instinctive direction for exploration.

Piotr Rudnicki, the other main MIZAR-C group member, made our study group complete. He has been a wise, experienced guide to my personal and professional philosophy—he mixed thoughtfulness with humor. Thanks go to my committee: Keith Smillie for his careful and appreciated evaluations and comments, and to Peter van Beek and Bruce Cockburn for their thorough and excellent review. I owe Brent Knight for his careful criticisms, and for insisting on excellence. I am also grateful to Tony Jones and Ladislav Hala whose work ethic in the weight room I have gained personal strength from. Ron Kube, Hazem Nassef, Manoj Jain, Phil Clarke, and Nyan Lo provided me with necessary dart challenges. I thank Bob Beck for allowing me to let loose, and the Comp Sci intramural hockey team for fun and support. Brett Manz deserves credit for being an excellent friend and for providing late-night/early-morning snooker competition and general diversions from reality. I thank my mother and father for their support and care, and for enduring my often nocturnal working schedule.

The Department of Computing Science and Jim Hoover have provided me with funding during my stay.

I owe Lorena the most, for her patience and love.

Contents

1	Introduction	1
1.1	Background Material	2
1.2	Programming using Constructive Proofs	3
1.3	The MIZAR-C System	6
2	Preliminaries	8
2.1	Scheme Notation for Untyped λ Calculus	8
2.2	Realization Pairing and Inference Notation	9
3	The Base Set of Inferences and its Soundness	11
3.1	Constructiveness and Marking Formulas	11
3.2	Familiar Inferences	12
3.2.1	Implication and Equivalence	13
3.2.2	Quantified Formulas	13
3.3	Existential Introduction and Elimination	14
3.3.1	Formula Strictness and Existential Introduction	15
3.3.2	The consider Elimination	16
3.4	Conjunction, Disjunction and Case Selection	19
3.4.1	Natural Conjunction and Disjunction Representation	20
3.4.2	Realizing Disjunctions	20
3.4.3	Disjunction Introduction	21
3.4.4	Case Selection	23
3.4.5	Conjunction Inferences	24
3.5	Logic with Equality	25
3.5.1	Equality Introduction Rule	26
3.5.2	Take Rule	26
3.5.3	Substitution Scheme	27
3.6	Sorted Variables	28
4	Frameworks for Relating Programs to Specifications	30
4.0.1	Shape	31
4.0.2	Adding Common Structures	32
4.0.3	Afunctional Realization Specification	34

4.0.4	Alternative Case Selection Strategies	36
5	Some Problems Encountered	39
5.1	An Example Proof	39
5.2	Implementation Restrictions	40
5.3	Deficiencies in the Logic	42
5.3.1	Sorts as Static Types	42
5.3.2	Tactics and Abbreviations	44
A	Annotated proof of Pred	50
B	Non-constructive Inference Rules	53
C	Proof of Summation	54
D	Proof of Summation using Tactics and Abbreviations	57

List of Figures

1.2.1 A demonstration proof of the identity function.	4
1.2.2 A program extracted from Ident 's proof.	4
1.2.3 An example proof of a function in MIZAR-C	5
3.3.1 A sample use of the consider statement.	17
3.3.2 Performing universal elimination on a term.	18
3.3.3 Using value capture for existential introduction.	19
4.0.1 A representation of a data type for the naturals.	32
4.0.2 Proving lemmas from a compact representation of a data type.	33
4.0.3 An example of parallel case selection.	38
5.3.1 An alternative grammar for sorts.	43
5.3.2 Using special syntax for sorts.	43
5.3.3 Using formulas to specify types.	44

List of Symbols

Notation	Description	Section
ρ	<i>realizes</i> : $x \rho Y$ means x realizes Y	2.2
$\langle el_0, el_1, \dots, el_n \rangle$	n element list containing el_i in order	2.1
$\vee \langle F_0, F_1, \dots, F_n \rangle$	n -ary disjunction in formulas F_i	3.4.1
$\wedge \langle F_0, F_1, \dots, F_n \rangle$	n -ary conjunction in formulas F_i	3.4.1
$\mathcal{D}(F)$	the set of defined terms in formula F	3.3.1
$\nu(v)$	the Scheme variable identifier to which the logic variable name v is mapped	3.2.2
$\mu(l)$	the Scheme variable identifier to which the formula label l is mapped	3.2.2
$A_{x \leftarrow t}$	the proper free substitution of term t for (free) variable x in A	3.2.2
$\bar{x}$	the Scheme expression x evaluated	2.1
$P \doteq F$	the predicate P abbreviates the formula F	5.1

Chapter 1

Introduction

There is interest in the computing science community in the relationships between computer programs, particularly functional programs, and proofs of theorems in various *constructive* logics. Some of that interest is in codifying mathematical thought using a formal logic for precision: the rigour with which computers can check formal proofs can be utilized to gain confidence in the proven theorems. This has been the concern of the AUTOMATH [7] project and the Mizar [21] project; the former using a constructive logic and the latter a classical logic.

Another important motivation for using a constructive logic is that constructive logics promise a method for producing correct programs. The main idea behind this kind of constructivism is that a given formula, F , is considered true when it can be proven constructively, that is, when there is some effective method for producing an object which gives evidence for F . It was Kleene (see [24, 2]) who proposed the idea of *realizability* which relates formulas to programs: the programs realizing formulas take the place of the vaguely defined idea of effective method of for giving evidence to the formulas. Curry and Howard (see [6]) reformulated realizability with natural deduction proofs in mind: they showed an isomorphism between natural deduction logics and typed lambda calculi, that is, they related propositions to types.

The relationship between propositions and types is used by projects whose concerns are directed towards program derivation. Martin-Löf's Type Theory [16], NUPRL [4], and the Calculus of Constructions [6] approach the problem of program derivation from the type perspective, using type systems as their formal language. In contrast, the **PX** system [10, 11] utilizes a logic whose term language forms the basis of a functional programming language; in **PX**, from a constructive proof of a formula, a Lisp program is extracted. Whereas **PX** implements a particular logic for programming, systems such as Elf [18] are *logical frameworks* in which various logics can be implemented, for example constructive logics. **PX** and Elf demonstrate a range of concerns for computing scientists: from investigating particular logics with a specific computation paradigm in mind, to investigating classes of logics in a framework for implementing.

The members of the MIZAR-C group at the University of Alberta, (Jim Hoover,

Piotr Rudnicki, and Andrew Walenstein) have implemented a proof checker and program extraction system, called MIZAR-C, which falls somewhere in the range between PX and Elf. MIZAR-C implements a first-order natural deduction logic which may be extended by the addition of inference rules and axioms. With the extension mechanisms, MIZAR-C can implement or simulate many different logics, so it is a type of logical framework. In addition, programs are generated from proofs through a realizability interpretation in which formulas and inferences in the logic calculus are interpreted as expressions and operations on expressions in an untyped lambda calculus. MIZAR-C's language for the logic is closely related to the Mizar-MSE language [14]; its language for the extracted programs is Scheme, a cousin of Lisp which implements an untyped lambda calculus. MIZAR-C's proof checking environment is implemented using the Synthesizer Generator [19] which provides an interactive syntax directed editing environment for writing proof texts.

1.1 Background Material

In order to comfortably read this thesis, knowledge of the general ideas of several systems and disciplines is desirable. The reader should understand the process of natural deduction, understand the untyped lambda calculus, and know how to read and understand the Scheme syntax for untyped lambda calculus. A good introduction to lambda calculus and Scheme appears in [8]; [17] and [14] give introductions to natural deduction, the latter also describing the syntax of Mizar-MSE, the language upon which MIZAR-C's language is based.

Natural deduction refers, in essence, to the process of using arguments to create formulas: an argument is created by making *assumptions* and showing a *conclusion* which follows from the assumptions. If the assumptions are formulas, the argument proves an implication: showing formula B can be concluded from an assumption of formula A proves $A \rightarrow B$. Similarly, if the assumption is a variable with an unknown value, the argument proves a universally quantified formula: showing formula $P[x]$ from a variable x of unknown value proves $\forall x.P[x]$. Forming implications this way is called *implication introduction*, and forming universally quantified formulas is called *universal introduction*. What is important for the logic is that when an implication or universal introduction is performed, an assumption is *discharged*, that is, we can no longer refer to the discharged assumption (A above), or variable (x above). In MIZAR-C, assumptions of formulas and variables occur only within the scope defined by a **now-end** pair, therefore universal and implication introductions occur when leaving the scope. Natural deduction systems, in addition to arguments, also define a collection of *rules of inference* such as *implication elimination*, which are used to justify deduction steps. For example, if A is known to be true, and we have proven $A \rightarrow B$, then we can use the implication elimination rule to justify the conclusion of B from these two facts. In MIZAR-C, justification of the deduction of formula F by appealing to an inference rule is written " F by rule($A, B, \dots$)", where $A, B, \dots$

are formulas used in the justification by rule *rule*. A natural deduction proof of a theorem typically utilizes a combination of inference rules and arguments.

In its simplest form, untyped lambda calculus is a calculus of variable substitution: variable substitution occurs when lambda calculus terms are *reduced*. Lambda expressions of the form `(lambda (x) g)` define a function in parameter variable *x* over expression *g*; lambda expressions of the form `(f e)` are called *applications* of expression *f* to argument *e*. If *f* is a function of the form `(lambda (x) g)`, `(f e)` can be *reduced*: the result of the reduction is the expression obtained by substituting all occurrences of expression *e* for variable *x* in expression *g*. This reduction is called β reduction. Scheme uses (an expanded version of) untyped lambda as its function definition and application language, so expressions of the form `(lambda (x) g)` are Scheme functions, and `(f e)` is a function application itself. Reduction occurs in Scheme during *evaluation*, and Scheme tries to reduce an expression until it cannot be reduced further. Scheme will do this by trying to evaluate (reduce) the parameters in a function application before reducing the function application. Scheme adds assignment operations to the language, however when these operations are not used, Scheme expressions are purely functional, that is, their output depends only on the value of their input parameters.

1.2 Programming using Constructive Proofs

MIZAR-C provides an environment for producing programs which are correct with respect to a specification of its intended behaviour. The relationship between specifications and programs is established by using the language of first order predicate logic formulas as the specification language. We can do this by relating input *x* to output *y* by reading the formula

$$\forall x.(\exists y.Post[x,y])$$

as a specification of a program which, when given an *x*, finds a *y* for which property *Post*[*x*,*y*] on *x* and *y* holds. Then, through the realization mechanism, from a constructive proof of the specification, a program is extracted which is correct with respect to the specification.

MIZAR-C uses a Curry-Howard translation to transform the proofs into Scheme programs. To see how the transformation relates proof steps to program steps, examine Figure 1.2.1. The figure contains a proof of the formula $\forall x.(\exists y.y = x)$, which is written in the language of MIZAR-C as "for x holds (ex y st y=x)". In the proof, the `let` statement declares a variable which is local to the scope delimited by the `now-end` pair. Since nothing is assumed about the value of the local variable *x*, whatever formula we conclude using it (in this case, `ex y st y=x`) will hold for any value of *x* which we provide; therefore, this natural deduction argument proves that `ex y st y=x` may be universally quantified.

```

now
  let x;                                { parameter x }
  x = x by eqintro;                     { Mizar-C knows x=x }
  thus ex y st y=x by exintro(_PREVIOUS); { return val == x }
end;

```

Ident: for x holds (ex y st y=x) by direct(_PREVIOUS);

Figure 1.2.1: A demonstration proof of the identity function.

```

(LAMBDA (x)
  (LIST x #t))

```

Figure 1.2.2: A program extracted from Ident's proof.

The use of local scopes for variables is a part of the process of hypothetical reasoning. In the logic, this reasoning corresponds to creating a universal formula (i.e. universal introduction), in programming terms, it corresponds to function abstraction. Thus the **let** variable declaration is analagous to a parameter declaration in a programming language. The existential quantifier introduction producing **ex y st y=x** acts as a value store: in this case the value can be seen to be the value of the local variable (function parameter) **x**. The **thus** token serves to identify **ex y st y=x** as the conclusion of the subproof, which corresponds to identifying the return value for a function in programming language terms. Therefore, a program extracted from the proof would look like the program in Figure 1.2.2. The program is a function which takes any value and returns it (along with a "proof" that the value is equal to itself, here written **#t**). In the lambda calculus, **(LAMBDA (x) e)** is a function abstraction over expression **e** using **x** as the parameter to the function.

Similarly, the **assume** statement is analagous to a precondition parameter: implication introduction corresponds to function abstraction, and implication elimination to function application. Implications abstract over formulas, so in our program-specification relationship, the corresponding function abstraction is over precondition verifications. In particular, functions realizing implications transform precondition witnesses to postcondition witnesses. This relationship is shown in the larger example proof in Figure 1.2.3. This small proof is of the existence of the predecessor function using the system extended with an implementation of natural numbers defining the constant 0 (**0**), the successor function (**succ**) and how it may be used, and an inference rule supporting weak induction for the naturals (**natind**). The formula that we prove, which we say is a specification for the predecessor function, is labelled **PredFunc**. The axioms required by the proof, which are provided by the **naturals-t** extension (not shown), are restated for clarity by using the **direct** rule, and are la-

```

article pred

environ
  extendwith naturals-t;

begin
  /* Restated axioms from naturals-t */
  AX-successor:
    for x st Nat[x] holds Nat[(succ x)] by direct(NatT-succ);
  AX-zero-nat:
    Nat[0] by direct(NatT-0);

  0=0 by eqintro();
  (succ 0)=0 or 0=0 by disjintro(_PREVIOUS);
  Nat[0] & ((succ 0)=0 or 0=0) by conj(_PREVIOUS, AX-zero-nat);
BaseCase:
  ex pred st Nat[pred] & ((succ pred)=0 or 0=0) by exintro(_PREVIOUS);

InductionStep:
  now
    let x;
    assume xNat: Nat[x];
    assume unusedIH: ex pred st Nat[pred] & ((succ pred)=x or x=0);

    Nat[x] implies Nat[(succ x)]      by univelim(AX-successor);
    Nat[(succ x)]                    by impelim(_PREVIOUS, xNat);
    (succ x) = (succ x)              by take(_PREVIOUS);
    (succ x) = (succ x) or (succ x)=0 by disjintro(_PREVIOUS);
    Nat[x] & ((succ x) = (succ x) or (succ x)=0)
      by conj(_PREVIOUS, xNat);
    thus ex pred st Nat[pred] & ((succ pred) = (succ x) or (succ x)=0)
      by exintro(_PREVIOUS);
  end;

PredFunc:
  for x st Nat[x] holds
    (ex pred st Nat[pred] & ((succ pred)=x or x=0))
    by natind(BaseCase, InductionStep);

```

Figure 1.2.3: An example proof of a function in MIZAR-C.

beled with names starting with the AX- prefix. Another version of this proof appears in Appendix A. In that version, the formulas and statements have been annotated with comments (delimited by /* and */) that describe the proof steps, and indicate the extracted program fragments which realize the formula immediately above them in the proof.

1.3 The MIZAR-C System

The fixed part of the proof calculus and interpretation defined by MIZAR-C is intentionally small. This serves to emphasize the use of the extension mechanisms of the system for exploring the features of different logical extensions. Fundamental features of logics, such as the inclusion of the Axiom of Choice, or the implementation of *data types* like natural numbers and lists, can therefore be defined and explored separately when implemented as extensions. Nevertheless, MIZAR-C is not a logical framework in the flavour of Elf, because its language for writing formulas and proofs is fixed. In addition, although MIZAR-C's metalanguage allows for the implementation of inference rules for many logics, MIZAR-C has not defined any particular metatheory, such as Elf's type theory, with which to expand MIZAR-C's basic logic.

MIZAR-C contains features which are important to proof derivation, and which distinguish it from other constructive logic and type systems:

1. MIZAR-C employs classical first order logic. Because of this feature, the proof writer is not confined to writing strictly constructive proofs in cases where constructions are not required. Classical inferences can be useful, for example, when they are used for proving the functional behaviour of a program: in order to soundly use the Axiom of Choice, a proof that the argument formula behaves functionally must be provided, and that proof does not need to be constructive. This feature distinguishes MIZAR-C from purely constructive logics and type theories, such as NUPRL, because these do not have natural analogues to non-constructive classical inferences (although corresponding procedures can be defined which simply remove constructive content from programs).
2. The term language, together with the inference rule design and interpretation, allow for a language of *partial* functions including functions not defined on the entire universe of discourse. Partial functions may play an important role in proof writing because they can be easier to use than total functions. In addition, a theory of manipulating partial functions is necessary, for example, in establishing extensions which encode a type-like system which uses the idea of "formulas as types". The inclusion of partial terms distinguishes MIZAR-C from Martin-Löf's Type Theory, which cannot represent partial terms.
3. Normally, in constructive logics, disjunctions are introduced with an introduction rule which ensures that only one of the disjuncts are true, that is, the cases

are disjoint. We interpret disjunctions as representing potentially non-disjoint cases by expanding the disjunction introduction rule and the disjunction realization scheme. This modification to standard interpretations leads to viewing disjunction elimination as simultaneous analysis of multiple cases instead of an (nested) if-then-else structure. When we have more than one case holding at one time we can then make use of parallel or non-deterministic execution during case selection in disjunction elimination to make choices for efficient computation. The author does not know of a system where disjunction may have non-disjoint cases, and where disjunction elimination may be realized by non-deterministic case analysis.

4. The logic can support an interpretation for existentially quantified formulas where the existential quantifier essentially captures what is a set of possible values rather than a single value. This interpretation allows formulas to specify relations which are realized by programs which behave in a manner which is not purely functional, that is, *afunctional*. Also, this type of interpretation is important in order to be able to represent results of the non-deterministic choice selection. Using *afunctional* realizations for existentially quantified formulas appears to be a possibility which is unexplored elsewhere, as the current research emphasizes purely functional executions exclusively.

After briefly introducing necessary notation in Chapter 2, we define in Chapter 3 the realizability interpretation for the logic and discuss the logic's soundness under the interpretation, including a discussion of our treatment of partial function terms. Then in Chapter 4 we discuss how our interpretation provides a partial framework for encoding programs in MIZAR-C's system and show alternate interpretations for case selection and its relationship to *afunctional* interpretations of existentially quantified formula. Then in Chapter 5 we present an example proof of a function, and we finish with a discussion of some problems with our implementation and weaknesses of the current system.

Chapter 2

Preliminaries

At each proof step, MIZAR-C's proof checker pairs formulas from the logic with expressions called *realizations* from another language called the *realization language*, under a realizability interpretation for the logic calculus. The realization language is based on the Scheme (cf. [3]) instantiation of the untyped lambda calculus (Scheme is a cousin of Lisp with similar syntax). We use the syntax of Scheme to denote the realizations of formulas with the anticipation that the reader will understand the Scheme notation or be able to translate it to any chosen representation. After introducing the important Scheme notation, we describe the ρ notation for denoting the pairing of realizations to formulas and show how we represent rules of inference.

2.1 Scheme Notation for Untyped λ Calculus

In order to write the constructions extracted from proofs, we make use of some Scheme special forms, some standard Scheme combinators, and we augment Scheme with some of our own combinators. The following is the essential Scheme syntax upon which we build:

- Lambda abstraction over a Scheme expression e is written as the special form (LAMBDA (x) e), where x is the abstracted variable. Application of the lambda form f to a Scheme expression e is written (f e).
- List construction appears as (LIST x_0 x_1 ... x_m) which reduces to the $m+1$ element list $\langle \overline{x_0}, \overline{x_1}, \dots, \overline{x_m} \rangle$, where $\overline{x_i}$ is the value of x_i evaluated in the current context.
- The expressions (CAR l) reduces to the first element in list l , and (CADR l) to the second element in l . Let $l = \langle l_0, l_1, \dots, l_m \rangle, m > 0$, and let $p = \langle p_0, p_1, \dots, p_n \rangle, n > 0$ where for $i \in \{0, \dots, n\}, 0 \leq p_i \leq m$. Then (PROJECT p l) reduces to the $n+1$ element list $r = \langle r_0, r_1, \dots, r_n \rangle$ such that $r_i = l_{p_i}$. That is, r is the projection of the elements in l according to the projection list p .

In following sections, while discussing the particular features of our logic, additional combinators and special forms will be introduced where required.

2.2 Realization Pairing and Inference Notation

In the following chapters we represent the pairing of a realization to a formula of the logic with the connective ρ . We write $x\rho Y$ when we mean x realizes Y , or more precisely, the Scheme expression x realizes logical formula Y . In addition, the following conventions will be used to denote logical formulas, concrete language sentences, and metalinguistic symbols:

- When we consider logical formulas, as for example in the statement of inference rules, they are written in the standard logical formula format such as $\forall x \in Nat. (\exists y. P)$ instead of the concrete syntax of MIZAR-C.
- Formulas are named with upper case letters, so for example the P in $\forall x. \exists y. P$ stands for any subformula which may contain x and y . General formulas can be distinguished from predicates by the presence of brackets: for example, the formula $P[x, y]$ stands for the binary predicate named P on x and y .
- When we are writing examples of formulas written in MIZAR-C syntax, they are written in typewriter style such as `for x st Nat[x] holds (ex y st P[x,y])`. In the sentence `(ex x st F)`, F is an arbitrary formula.
- Scheme expressions are also written in typewriter style such as `(LAMBDA (x) x)`.

We note that it is only the realization extraction machinery which remembers the pairing of the realization to the logic formula, for example, two different interpretations and their extraction mechanisms may produce different programs from the same proof. Since the proof writer only writes proofs, not extractions, proof writer need not see the extractions and they are usually hidden. Thus $x\rho Y$ is a phrase in the metalanguage and when it appears in place of a formula the reader should understand that Y is the formula involved and x is the realization maintained by the interpretation machinery.

It is essential to remember that the lambda calculus places the same importance on scope as the logic, therefore it should be understood that we write $x\rho Y$ as a short form for "expression x in context Γ_R realizes formula Y in context Γ_L ", where Γ_R is the Scheme environment corresponding to the logic context Γ_L in which Y appears. This consideration is important when we discuss the soundness of the interpretation of quantified formulas and implication, both of which use Scheme special forms to create environments which assign realizations (Scheme expressions) to symbols.

We write our inference rules in a style adopted from Genzen's natural deduction rule style (as in [24]) with the understanding that it is a mechanical although tedious exercise to translate our object language (which uses labels to reference multiple

formulas) into Genzen style proof trees. As indicated above, we take the liberty of writing metalinguistic phrases like $x\rho Y$ instead of simply the formula Y in order to indicate how extraction is performed during the interpretation of the inferences.

As an illustration, we write the imaginary inference rule *IR* as:

$$IR \frac{\Gamma \vdash a_0 \rho A_0 \quad \Delta \vdash a_m \rho A_m}{\Theta \vdash b \rho B} .$$

Sequents are used for our rules because, in following discussions, we will want to be able to explicitly mention that certain variables are available when performing hypothetical reasonings. When a formula F is inferred from the formulas Γ when there is a variable v in the context, we write $\Gamma \vdash F$. To avoid distractions, our rules are written with a common formula sequence, such as Γ , as the sequence on the left hand side of the turnstile (' $\vdash$ ') instead of the more general form which uses possibly distinct sequences Δ and Θ . The ambitious reader can restate formulas contained in the sequences Γ , Δ , and Θ in a use of *IR* above to make them equal and then *thin* the assumptions at the top level (as described in [17]).

In addition, where a hypothetical reasoning (an unnamed *now* reasoning) is actually used to perform the inference in question, such as in implication introduction, the common names such as $\rightarrow$ -*intro* will be used to identify these deductions. Otherwise we will use the actual inference scheme names as implemented in MIZAR-C, such as *disjintro*. In all cases, we leave it up to the reader to perform the translation from the concrete object language sentences of MIZAR-C to the formula metalanguage we use.

We make a distinction between an inference rule and a *tactic*. A tactic is similar to a derived rule of inference: it is defined by a fixed sequence of deductions using (the primitive) inference rules. For example, if F is an arbitrary formula, the deduction

$$\begin{array}{l} \text{univelim} \frac{\Gamma \vdash u \rho \forall x. (\forall y. F)}{\Gamma \vdash (u \ \nu(a)) \rho \forall y. F_{x \leftarrow a}} \\ \text{univelim} \frac{\Gamma \vdash (u \ \nu(a)) \rho \forall y. F_{x \leftarrow a}}{\Gamma \vdash ((u \ \nu(a)) \ \nu(b)) \rho (F_{x \leftarrow a})_{y \leftarrow b}} \end{array}$$

on variables a and b , whose deduction steps are two uses of the *univelim* rule, might be called the tactic *dunivelim* (see section 3.2.2 for the meaning of μ and $F_{x \leftarrow a}$). Tactic *dunivelim* could be written as:

$$dunivelim \frac{\Gamma \vdash u \rho \forall x. (\forall y. F)}{\Gamma \vdash ((u \ \nu(a)) \ \nu(b)) \rho (F_{x \leftarrow a})_{y \leftarrow b}} .$$

Chapter 3

The Base Set of Inferences and its Soundness

In this chapter we present the *base* set of inferences, that is, the fixed set of inference rules defined by MIZAR-C. We also present a partial verification of their soundness with respect to the *base interpretation*: the interpretation of the base set of inferences. The logical calculus contains classical inference rules, so we mean soundness with respect to *constructive proofs*. Thus we say the interpretation is sound if, whenever a theorem is proven constructively, a program is extracted which is correct with respect to the specification as represented in the proven theorem. Furthermore, we show that for any proof, we can determine whether we can interpret the proof as being constructive or not so that we know if we have a valid extraction or not. For this we need a method of identifying constructive proofs; we show in section 3.1 that this identification can be performed inductively by *marking* each inference (proof step).

Some inferences are very familiar and are discussed in many other sources so we tersely present them and their interpretation in section 3.2. We instead concentrate on harmonizing the interpretation with the special features of the logic: existential elimination by use of the **consider** statement, *n*-ary conjunction and disjunction, equality manipulation schemes, partial functions, and sorted variables.

3.1 Constructiveness and Marking Formulas

Including classical inference rules into the logic requires us to introduce a method of marking formulas to indicate if their proof was constructive or not, in order to know if we have extracted a correct program. To make the marking notation unobtrusive we overload the ρ notation by having it specifying a marking as well as indicating a pairing. Thus we say formula Y is *marked as constructive* when we have written $x\rho Y$, that is, Y has been constructively arrived at. Similarly, we say formula Y is marked as *non-constructive* if Y either appears with the special Scheme constant **#f** as **#f** ρY or, simply as Y .

When justifying an inference by reference to a rule, every premise (upper sequent)

marked constructive is required to be satisfied by a formula marked constructive. For example, in an imaginary inference rule

$$\text{rule } \frac{\Gamma \vdash a\rho A \quad \Gamma \vdash B}{\Gamma \vdash c\rho C},$$

the premise (upper sequent) $\Gamma \vdash a\rho A$ is marked constructive and must be satisfied by a formula also marked constructive, whereas the premise $\Gamma \vdash B$ is marked non-constructive and may be satisfied by a formula which is arbitrarily marked.

In the practice of writing proofs, it may happen that a premise marked constructive may fail to be satisfied by a formula marked constructive. In this case, although a constructively marked conclusion is supported by the rule, we say a *construction error* has occurred. When a construction error occurs, the conclusion is always marked non-constructive regardless of the statement of the rule. For example, if *rule* were used where the premise $a\rho A$ was satisfied with a formula marked non-constructive, a construction error has occurred and the realization of the conclusion, c , is $\#f$. The definition of the marking scheme is completed by requiring that all axioms and assumptions be marked constructive by default.

There is a class of inferences in the base logic which do not require any of their premises to be marked constructive and have a conclusion marked non-constructive. These inferences we call *inherently non-constructive* for our interpretation, or simply *non-constructive*, as opposed to the inferences which provide an expression which realizes the conclusion if the marking of the premises are correctly met. For reference, the non-constructive rules appear in Appendix B. The remainder of the inferences in the base set are all potentially constructive—all of their conclusions are marked constructive. In each of these, if their premises are satisfied without a construction error, the conclusion is marked constructive and the expression realizing the conclusion is the realization extracted from the inference. By marking the formulas in all derivations, we know by induction on the marking of the inference rules that formulas marked constructive correspond exactly to those theorems for which correct programs have been extracted. We can state this property of the marking scheme as follows:

Property 3.1 From assumption set Γ , if we can deduce F , that is if $\Gamma \vdash r\rho F$, then $r \neq \#f$ and F is marked constructive iff the proof of F was constructive.

Proof By induction on the length of the derivation, using the definition of the marking scheme.

3.2 Familiar Inferences

The following inferences and their realizations closely follow the standard Curry-Howard translation; we merely present them here.

3.2.1 Implication and Equivalence

$$\rightarrow\text{-intro} \frac{\Gamma, A \vdash b \rho B}{\Gamma \vdash (\text{LAMBDA } (\mu(A)) \ b) \rho A \rightarrow B}$$

$$\text{impelim} \frac{\Gamma \vdash i \rho A \rightarrow B \quad \Gamma \vdash a \rho A}{\Gamma \vdash (i \ a) \rho B}$$

Here $\mu(A)$ is the Scheme variable name to which the label used to identify assumption of A in the deduction of $\Gamma, A \vdash B$ in $\rightarrow\text{-intro}$ is mapped. From the definition of the lambda form, $\mu(A)$ binds the realization of A , namely a , on application in impelim . In our implementation we ensure that the $\mu(A)$ created in $(\text{LAMBDA } (\mu(A)) \ b)$ is unused in order to avoid capturing any Scheme variables in b . $\mu(A)$ is analogous to the *realizing variables* in [23]. Since $\mu(A)$ represents the realizing variable for an assumption, the expression it is bound to must be of the appropriate shape, as described in section 4.0.1.

The biconditional, $\leftrightarrow$, is defined as a conjunction of two implications, so iffintro and iffelim could be implemented as tactics, however, because MIZAR-C does not have support for reasoning about tactics, we implement them as the rules:

$$\text{iffintro} \frac{\Gamma \vdash l \rho A \rightarrow B \quad \Gamma \vdash r \rho B \rightarrow A}{\Gamma \vdash (\text{LIST } l \ r) \rho A \leftrightarrow B}$$

$$\text{iffelim (left)} \frac{\Gamma \vdash i \rho A \leftrightarrow B}{\Gamma \vdash (\text{CAR } i) \rho A \rightarrow B} \quad \text{iffelim (right)} \frac{\Gamma \vdash i \rho A \leftrightarrow B}{\Gamma \vdash (\text{CADR } i) \rho B \rightarrow A}$$

The $\leftrightarrow$ rules pair and unpair realizations for implications, so it is clear that to properly unpair we must be careful about the order in which the realizations are paired. In particular, for any $A \leftrightarrow B$ it is not true that if $r \rho A \leftrightarrow B$ then $r \rho B \leftrightarrow A$, although from one formula the other can easily be proved. Thus we postulate the tactic iffswap which performs the required proof, which we have currently added as the rule:

$$\text{iffswap} \frac{\Gamma \vdash i \rho A \leftrightarrow B}{\Gamma \vdash (\text{PROJECT } (\text{LIST } 1 \ 0) \ i) \rho B \leftrightarrow A}$$

3.2.2 Quantified Formulas

To interpret quantified formulas, we must map variable identifiers from the logic onto Scheme variable identifiers. Define a mapping ν such that for a given defined logic variable identifier x , the Scheme identifier it is mapped to is $\nu(x)$, such that distinct logic variable identifiers are mapped onto distinct Scheme variable identifiers. *Complex* terms in the logic borrow their syntax directly from Scheme syntax: a well formed complex term is of the form $(f \ t_0 \ t_1 \ \dots \ t_n)$, $n > 0$ where f is a variable and t_i are all variables or well formed complex terms. Then, for the well formed complex term

$t \equiv (f\ t_0\ t_1\ \dots\ t_n), n > 0$, we can simply define $\nu(t) \equiv (\nu(f)\ \nu(t_0)\ \nu(t_1)\ \dots\ \nu(t_n))$. Expanding the ρ notation to allow realizations to be paired with terms as well as formulas, we write $\nu(t)\rho\ t$ if

- t is a well formed term which is defined in the current context (see section 3.3.1), and
- the Scheme variables identified by $\nu(t)$ are bound appropriately so that the value of $\nu(t)$ implements t in any formula in which t appears.

Recall that the correct bindings of the Scheme variables are guaranteed by the interpretation of universal quantification as lambda abstraction and universal elimination as application. Thus the rules for universal quantifiers are

$$\forall\text{-intro} \frac{\Gamma^x \vdash a\rho A}{\Gamma \vdash (\text{LAMBDA } (\nu(x))\ a)\rho \forall x.A} \quad \forall\text{-elim} \frac{\Delta^y \vdash f\rho \forall x.B}{\Delta^y \vdash (f\ \nu(y))\rho B_{x \leftarrow y}}$$

where A and B are formulas in free x , and where x and y are variables in Γ^x and Δ^y respectively. Γ^x represents the context Γ with a variable x added to it, and Δ^y represents a context Δ where the variable y is defined. The variable x is not defined in Γ so we say it is a variable *local* to the reasoning which establishes $\Gamma^x \vdash a\rho A$. Also, $\nu(x)$ and $\nu(y)$ are the Scheme variable identifiers which the variables x and y are mapped to in Γ^x and Δ^y respectively. $B_{x \leftarrow y}$ is the formula B with all occurrences of the free variable x replaced by the variable y .

The relationship between the quantified logical formula and the extracted Scheme expression which establishes the soundness of the interpretation is preserved by our construction. This is commonly justified by observing that the substitution performed in the β reduction of $(f\ \nu(y))$ corresponds to the bound variable substitution in establishing $B_{x \leftarrow y}$. Likewise, the lambda abstraction over all occurrences of the Scheme variable $\nu(x)$ in a corresponds to the universal quantification over x all occurrences of x in A .

3.3 Existential Introduction and Elimination

MIZAR-C allows for the formation of partial functions by allowing the formation of complex terms which may not denote values in all contexts. In addition, we do not have any explicit type checking for variables which form a term, so it is possible to form terms which are not defined when it is valid to introduce arbitrary formulas. Arbitrary formulas may be introduced in disjunction introduction or in the assumptions during hypothetical reasonings. In order to avoid problems from either of these cases where terms may not have any meaning, we require that whenever we use an inference rule which considers a term's value, such as existential introduction, we need to be assured that the inference is allowed only when the term is defined. In the following section we introduce a notion of when the logic ensures that a term must be defined; this leads directly to the existential introduction rule.

3.3.1 Formula Strictness and Existential Introduction

We do not have a special predicate or notation to explicitly indicate that for a complex term t , it has a defined value (which, for example, Beeson in [2] writes as ' $t\downarrow$ '). Because of this, in disjunction introduction (see section 3.4.3), we cannot require that all complex terms in the disjuncts added by the introduction be explicitly indicated as being defined since explicit indication is not possible. In addition, type-checking for complex terms is not available in the current version of MIZAR-C so that any well formed complex term may be written in the disjuncts introduced during disjunction introduction. Similarly, for hypothetical reasonings which introduce implications (see section 3.2), we do not check the assumption to ensure that all complex terms in it are defined. Notice that even though formulas may contain complex terms which may not be defined, the system checks that all simple terms (variables) occurring in the assumption or disjuncts are defined.

In order to ensure that these arbitrary additions do not cause trouble, we show that the logic indicates exactly when a complex term must be defined based on the notion of *derivability* of atomic formulas. The result follows from a strictness requirement for the terms in atomic formulas: any atomic formula which has been derived in any context must be *fully defined* in that context. In a fully defined formula, all terms and subterms occurring in it must be defined in the current context. Clearly, for any atomic formula which was proven, all of its terms and subterms must be defined in order for the derivation to make sense and the formula to be considered true. Since all subterms in defined complex terms are required to be defined, the term language does not support special forms or control structures (such as a partially evaluated *if* form) in the term expressions.

Thus a fully defined formula contains a set of complex terms which are themselves fully defined. For a formula F this set is written as $\mathcal{D}(F)$ and we can define it for all formulas based on the notion of the derivability of a fully defined formula. Let $\mathcal{D}(F)$ be defined as:

1. If $F = \vee \langle F_0, F_1, \dots, F_m \rangle, m > 0$, $F = A \rightarrow B$, or $F = \neg A$ for formulas F, F_i, A , and B , then $\mathcal{D}(F) = \emptyset$.
2. If F is a predicate or equality, then it must be fully defined so all complex terms in F must denote values for F to be derived; therefore $\mathcal{D}(F)$ is the set of all complex terms, including subterms, which occur F . All subterms in any complex term must also be defined in order for the term to be defined.
3. If $F = \wedge \langle F_0, F_1, \dots, F_m \rangle, m > 0$ is a conjunction, any of the conjuncts F_i must be derivable and fully defined in the context so $\mathcal{D}(F) = \cup_{i=0}^m \mathcal{D}(F_i)$.
4. If $F = \forall x.P$, or $F = \exists x.P$ for some formula P , then $\mathcal{D}(F)$ is the subset all the complex terms in $\mathcal{D}(P)$ which do not contain the free variable x which is bound in F .

Proposition 3.3.1 If t is a complex term, $t \in \mathcal{D}(F)$ only if t is defined in the current context.

Proof The proof is by showing for each case of the definition of $\mathcal{D}$, no complex terms are added to $\mathcal{D}(F)$ which are not defined in the current context. Justification of cases 2 and 3 is straightforward as a result of the strictness requirement for atomic formulas and the meaning of conjunction. Case 4 is justified by considering the possibility of using universal or existential elimination on F . For either case, it would be impossible to perform universal or existential elimination to derive the formula P such that it is fully defined, if $\mathcal{D}(P)$ is nonempty but $\mathcal{D}(F)$ is empty. Therefore the complex terms available at the level in which F is stated are those which are defined in P without reference to the quantified variable x . No terms are added to $\mathcal{D}(F)$ in case 1.

Note that case 1 indicates that we cannot decide if a disjunction, implication, or negation asserts that any complex term in it is defined. Since both the assumption in hypothetical reasoning, and disjunction introduction can produce subformulas which have undefined complex terms, without a proof history we cannot decide which, if any, of the complex terms in either of these formulas are defined. Finally, $\neg A$ may be identified with the formula $A \rightarrow \perp$, so we treat negated formulas as if they were implications; that is we cannot trust that the complex terms in $\neg A$ are defined.

With the above definition of $\mathcal{D}(F)$, and the fact that MIZAR-C does not permit the statement of formulas with undefined variables in them, we have an understanding of when a term in a formula F denotes a value. Therefore we may now state the existential introduction rule:

$$\text{exintro} \frac{\Gamma \vdash u \rho A_{x \leftarrow t}}{\Gamma \vdash (\text{LIST } \nu(t) \ u) \rho \exists x.A},$$

where A contains free x in it (the formula may not be vacuously quantified), and t is either a variable, or $t \in \mathcal{D}(A_{x \leftarrow t})$. If $A_{x \leftarrow t}$ is marked constructive, for all terms $t = (f \ t_0 \ t_1 \ \dots \ t_m)$, $m > 0$, such that $t \in \mathcal{D}(A_{x \leftarrow t})$, $\nu(t) \rho t$ (from the interpretation of section 3.2.2). The type of Scheme expression which realizes $\exists x.A$ defined by this introduction rule is a list l where $(\text{CAR } l) \rho t$ and where $(\text{CADR } l) \rho A_{x \leftarrow t}$. With this observation, by explicit substitution it is clear that if $\nu(x)$ is bound to $(\text{CAR } l)$ (i.e. $\nu(t)$) then $(\text{CADR } l) \rho A$ also. That particular consequence is exploited by the existential elimination inference described in the following section.

3.3.2 The consider Elimination

MIZAR-C contains a convenient method of performing existential elimination which simplifies reasoning about existentially quantified formulas. In MIZAR-C, an elimination can be performed on an existentially quantified formula $\exists x.P$ by the action of considering a value of x which makes P true. Consideration of such a value is justified

```

now
  assume U: for x holds P[x] implies Q[x];
  assume E: ex y st P[y];
  now
    consider z such that EL: P[z] by direct(E);
    P[z] implies Q[z] by univelim(U);
    Q[z] by impelim(_PREVIOUS,EL);
    thus ex t st Q[t] by exintro(_PREVIOUS);
  end;
  thus ex t st Q[t] by direct(_PREVIOUS);
end;
(for x holds P[x] implies Q[x]) implies
  ((ex y st P[y])) implies (ex t st Q[t])) by direct(_PREVIOUS);

```

Figure 3.3.1: A sample use of the **consider** statement.

in the case where $\exists x.P$ is constructively proven and $\exists x.P$ is not vacuously quantified because, as the previous section demonstrated, the Scheme expression realizing such a formula $\exists x.P$ remembers the realization for a value of x which establishes P . The elimination is performed under the direction of the **consider** statement, and we give an example of its use in Figure 3.3.1. In the example, the variable z is said to be *local* to the context after the **consider** statement. For clarity, the **consider** statement and the scope of the local variable z are delimited by an optional set of **now** and **end** delimiters.

The format of all deductions which the **consider** statement allows can be represented by the following deduction sequence:

$$\text{consider} \quad \frac{\Gamma \vdash e\rho \exists x.P}{\frac{\Gamma^y \vdash m\rho P_{x \leftarrow y}}{\frac{\Gamma^y \vdash t\rho Q}{\Gamma \vdash f\rho Q}}} \quad (*)$$

where $P_{x \leftarrow y}$ is the formula P with all occurrences of variable x replaced with variable y which is local to context Γ as indicated by Γ^y (the fact that y is a local variable is the only distinction between Γ and Γ^y). Γ^y and $m\rho P_{x \leftarrow y}$ should be thought of as the scope and formula respectively which result from the use of the **consider** statement eliminating the existential quantifier from $\exists x.P$ as described above. m will be **#f** if $\exists x.P$ is marked non-constructive. The double line between $m\rho P_{x \leftarrow y}$ and $\Gamma^y \vdash t\rho Q$ indicates that a series of deductions may be performed in order to obtain $t\rho Q$ so long as the deductions do not add any more undischarged assumptions. The final step from $\Gamma^y \vdash t\rho Q$ to $\Gamma \vdash f\rho Q$ indicates that Q does not contain the local variable y introduced at the **consider** statement. MIZAR-C's proof checker explicitly verifies that Q does not contain references to the local variable although our sugared syntax of

```

now
  assume Express: P[(t u)];
  assume U: for x holds Q[x];
  (t u) = (t u)    by take(Express);
  consider v such that 1: v=(t u)  by exintro(_PREVIOUS);
  Q[v]            by univelim(U);
  thus Q[(t u)]   by equality(_PREVIOUS,1);
end;

```

Figure 3.3.2: Performing universal elimination on a term.

syntax of reasonings sometimes obscures where this check occurs. In Figure 3.3.1, the formula Q appears as $\text{ex t st } Q[t]$ and $\exists x.P$ appears as $\text{ex t st } P[t]$.

For our interpretation of the y considered to establish P , we bind $\nu(y)$ to the value previously established in the construction of $\exists x.P$, that is, $(\text{CAR } e)$ where $e \rho \exists x.P$. Then, as indicated in the existential introduction section 3.3, m is $(\text{CADR } e)$. The mechanism for establishing the binding of $\nu(y)$ and m appears in f as the expression

$$f \equiv (\text{LET } ((\nu(y) (\text{CAR } e))) (\text{LET } ((\mu(P) (\text{CADR } e))) t))$$

where $\mu(P)$ is the Scheme (realizer) variable corresponding to the label for the considered formula $P_{x \rightarrow y}$ at $(*)$ and $t \rho Q$ in the scope Γ^v .

In logical systems which cannot introduce new local variables, the above deduction can be transformed into the use of another common formulation of existential elimination, $\exists$ -elim:

$$\begin{array}{c}
 \rightarrow\text{-intro} \quad \frac{\Gamma^v, g \rho P_{x \rightarrow y} \vdash t \rho Q}{\Gamma^v \vdash (\text{LAMBDA } (\mu(P)) t) \rho P_{x \rightarrow y} \rightarrow Q} \\
 \forall\text{-intro} \quad \frac{\Gamma \vdash (\text{LAMBDA } (\nu(y)) s) \rho \forall y. P_{x \rightarrow y} \rightarrow Q}{\Gamma \vdash (\text{LAMBDA } (\nu(y)) s) \rho \forall y. P_{x \rightarrow y} \rightarrow Q} \quad \Gamma \vdash e \rho \exists x P \\
 \exists\text{-elim} \quad \frac{\Gamma \vdash (\text{LAMBDA } (\nu(y)) s) \rho \forall y. P_{x \rightarrow y} \rightarrow Q \quad \Gamma \vdash e \rho \exists x P}{\Gamma \vdash ((u (\text{CAR } e)) (\text{CADR } e)) \rho Q}
 \end{array}$$

where y is a variable, u is $(\text{LAMBDA } (\nu(y)) s)$ and s is $(\text{LAMBDA } (\mu(P)) t)$ respectively. In Scheme the special form $(\text{LET } ((a b)) c)$ is by definition equivalent to $((\text{LAMBDA } (a) c) b)$. Therefore by simple transformation it can be seen that the realization defined by f above and $((u (\text{CAR } e)) (\text{CADR } e))$ are equivalent Scheme expressions.

With the statement of existential introduction and elimination given, we can now show that the restriction of universal introduction and elimination to operating only over variables is not an important restriction. It can be shown that universal elimination on terms can be carried out in multiple steps by making use of local variables which denote terms. We start with an example of the procedure which is given in Figure 3.3.2. In it, the formula labeled **Express** is a formula containing a defined function term $(t u)$ with which we eliminated the universal quantifier in the formula at U .

```

now
  assume D: OK[(t x)];
  1: OK[(t x)] or BAD[(t y)] by disjintro(D);
  (t x) = (t x)    by take(D);
  consider v such that 2: v=(t x) by exintro(_PREVIOUS);
  OK[v] or BAD[(t y)] by equality(1,2);
  thus ex v st OK[v] or BAD[(t y)] by exintro(_PREVIOUS);
end;

```

Figure 3.3.3: Using value capture for existential introduction.

As we will show in section 3.5.2, if a complex term t is defined in a context we may prove that $t = t$ by using the **take** rule. With $t = t$ we can create a local variable by considering the term in the equality by using existential introduction rule as shown at label 1 above. After introducing the local variable, the universal elimination can be performed on it and all occurrences of the variable can be substituted with the term using equality substitution. The technique of introducing a local variable which temporarily names a defined term is called *value capture*. The above strategy works for any complex term and any universally quantified formula, so it can in fact be implemented as a tactic on basic inference rules.

Also, using the value capture strategy and equality substitution, we can prove existentially quantified formula over disjunctions created by disjunction introduction as is exemplified by the proof fragment in Figure 3.3.3. In the proof, $(t\ x)$ represents a defined complex term and $(t\ y)$ represents a possibly undefined complex term which was introduced using disjunction introduction. Like the strategy for eliminating universal quantifiers by substituting in for a term, the above strategy for introducing existentials is a tactic which will work for any disjunction introduction sequence from a formula with a defined term in it.

3.4 Conjunction, Disjunction and Case Selection

Since we intend to use MIZAR-C to write and check proofs, we are willing to introduce some features in the logic which may make reasoning about metalogical features more difficult than they could be, so long as some gains are made in concerns of the language interface. We show in the following sections how we formulate disjunction, conjunction, and case analysis in an intuitive way, but wait until the next Chapter to show how it is possible that this format of disjunction is more significant than merely syntactic sugaring.

3.4.1 Natural Conjunction and Disjunction Representation

In many other constructive logic and type theory systems, disjunctions are commonly defined as a binary connective (cf. [12, 24] or [22]). However, conjunction and disjunction are naturally viewed as operators upon sequences of formulas rather than a connective between a pair of formulas. As an illustration, consider that even if disjunction is defined to be a binary connective, the formula $A \vee B \vee C$ is commonly read as “at least one of A , B , or C is true” instead of the stricter readings of “one of A or $B \vee C$ is true” or “one of $A \vee B$ or C is true” (depending on $\vee$ ’s left-to-right precedence). More specifically, syntactically and realization-wise $(A \vee B) \vee C$ and $A \vee (B \vee C)$ are not equivalent although they are logically equal. Even so, we might define a tactic to shuffle the order of the connection among the disjuncts should we ever need to write a disjunction in a distinct but equal form. However, this raises the question of why one needs a series of deductions (a tactic) to prove two formulas are the same when the user intuitively sees them as equivalent?

Thus there is good reason then to *define*, as we have, $A_0 \vee A_1 \dots \vee A_n, n > 0$ to be infix notation for the generalized n -way disjunction $\vee \langle A_0, A_1, \dots, A_n \rangle$ on formulas A_i . We have defined conjunction of formulas similarly so that the conjunction written as $A_0 \wedge A_1 \wedge \dots \wedge A_m, m > 0$ is defined to be the infix notation for the m -way conjunction $\wedge \langle A_0, A_1, \dots, A_m \rangle$.

3.4.2 Realizing Disjunctions

We step back for a moment from examining inferences and discuss a version of the Kleene realizability for disjunction (cf. [24]) inspired by Martin-Löf (as in Backhouse, et. al [20]):

$$\begin{aligned} \text{inl}(x) \rho A \vee B & \text{ if } x \rho A \quad \text{or} \\ \text{inr}(x) \rho A \vee B & \text{ if } x \rho B \end{aligned}$$

where *inl* and *inr* are the “inject left” and “inject right” functions. We can imagine that the result of *inl*(x) is the sequence $\langle x, \text{left} \rangle$ and of *inr*(x) is the sequence $\langle \text{right}, x \rangle$ for language constants *left*, *right*. Then $\langle x, \text{left} \rangle \rho A \vee B$ means $A \vee B$ was proven from A , and $\langle \text{right}, x \rangle \rho A \vee B$ means $A \vee B$ was proven from B . Although this definition for disjunction has the advantage that it is straightforward (as are the inference rules for manipulating these disjunctions), it does not capture our understanding of disjunctions as being an operation on sequences of formulas. Furthermore, this formulation has the property that for any disjunction, the extracted realization remembers the extracted program of exactly one disjunct in a disjunction.

We can reformulate the above idea of realizability in order to capture the intuitive list-operator notion of disjunction. Let

$$\langle x_0, x_1, \dots, x_m \rangle \rho \vee \langle A_0, A_1, \dots, A_m \rangle, m > 0$$

if there is an $i \in \{0, 1, \dots, m\}$ such that $x_i \rho A_i$ and A_i is true, and for every $i \in \{0, 1, \dots, m\}$, either $x_i \neq \#f$ or else $x_i \rho A_i$ and A_i is true. In essence, a list

$\langle x_0, x_1, \dots, x_m \rangle$ realizes an m -formula disjunction if at least one of x_i realizes the corresponding disjunct A_i , and all the other x_i are the constant $\#f$ or else realize their corresponding (true) disjunct, A_i . This formulation of the realizability of the disjunction is closely related to the intuitive understanding of the disjunction; the property of “at least one of the formulas is true” is clearly similar to “at least one of the formulas is realized”.

$\#f$ is used in the interpretation in place of the evocatively named constants *left* and *right* since both *left* and *right* may be replaced by $\#f$. This is because *left* and *right* both indicate the placement of the other expressions realizing the true disjuncts by virtue of their position in the sequence (and by being an invalid realization), rather than by their distinct names. Since we are assuming $\#f$ to be a special language constant, we can aid our analysis of the soundness of our rules in the following section by insisting on the following properties of the extracted realizations in general:

Property 3.4.2.a No constructively proven formula is allowed to be realized by $\#f$.

Property 3.4.2.b All non-constructively proven formula must be realized by $\#f$.

Property 3.4.2.a can be proven inductively over the length of the derivations based on the realization extraction mechanisms for each inference and the appropriate requirements imposed on axiom realizations (see section 4.0.1). In order to show Property 3.4.2.b we need to appeal to the marking scheme described in section 3.1 which ensures that non-constructively proven formulas are realized by $\#f$. With these properties for the realizations for all formulas, we can define a marking for disjunction introduction which both preserves properties 3.4.2.a and 3.4.2.b and tries to ensure that *all* those x_i , for which the corresponding A_i was constructively proven, have realizations which are non- $\#f$.

3.4.3 Disjunction Introduction

An obvious generalization of the standard binary introduction is

$$\vee\text{-intro} \frac{\Gamma \vdash a_k \rho A_k}{\Gamma \vdash (\text{LIST } a_0 \ a_1 \ \dots \ a_m) \rho \vee \langle A_0, A_1, \dots, A_m \rangle} \quad m > 0, 0 \leq k \leq m$$

where $a_i = \#f$ iff $i \neq k$. This inference would give us a base inference rule set as powerful as the system with binary disjunction and the more restricted introduction rule; its interpretation preserves the soundness. However, the rule extracts an expression which remembers the realization of only a single disjunct. In order to advantageously use the case analysis described in section 3.4.4 (and especially in section 4.0.4), the inference must be modified to allow for more than one disjunct to be true.

The key idea is to allow other formulas to contribute their constructive proofs to the proof of the disjunction. Intuition tells us that disjunction is a weaker statement than conjunction, so we could think of allowing $\vee \langle A_0, A_1, \dots, A_m \rangle, m > 0$ to be concluded from $\wedge \langle A_0, A_1, \dots, A_m \rangle$. Instead, we can more directly conclude it from

the statements of the A_i themselves to obtain the rule $\vee\text{-intro}_f$ which modifies $\vee\text{-intro}$ as follows:

$$\vee\text{-intro}_f \frac{\Gamma \vdash a_0 \rho A_0 \quad \Gamma \vdash a_1 \rho A_1 \quad \dots \quad \Gamma \vdash a_n \rho A_n}{\Gamma \vdash r \rho \vee \langle B_0, B_1, \dots, B_m \rangle} \quad n \geq 0, m > n$$

where there exists a total 1-1 mapping $P : \{0, 1, \dots, n\} \mapsto \{0, 1, \dots, m\}$, such that for all $i \in \{0, 1, \dots, n\}$, $A_i = B_{P(i)}$. Then define an r' such that $r' \rho \vee \langle B_0, B_1, \dots, B_m \rangle$ so that by our discussion in section 3.4.2, $r' \equiv \langle r_0, r_1, \dots, r_m \rangle$ where

$$\text{for all } i \in \{0, 1, \dots, m\}, \begin{cases} r'_i \rho A_j & \text{if } r'_i \neq \#f \text{ and there exists } j \in \{0, 1, \dots, n\} \\ & \text{such that } A_i = B_{P(j)} \\ r'_i = \#f & \text{otherwise} \end{cases}$$

In other words, the extracted forms a_i which realize A_i are projected onto the list which realizes the disjunction.

In order to create r' by r , we define

$$r \equiv (\text{PROJECT } p \text{ (LIST } a_0 \ a_1 \ \dots \ a_m \ \#f))$$

where $p \equiv (\text{LIST } p_0 \ p_1 \ \dots \ p_m)$ and

$$\text{for every } i \in \{0, 1, \dots, m\}, p_i \text{ is } \begin{cases} k & \text{if } i = P(k) \text{ for some } k \in \{0, 1, \dots, n\}, \\ m + 1 & \text{otherwise.} \end{cases}$$

By the definition of PROJECT, it is straightforward to see that r reduces to r' as above, so the inference along with caseanal (see section 3.4.4) preserves the soundness of the interpretation since it satisfies the properties 3.4.2.a and 3.4.2.b.

As $\vee\text{-intro}_f$ is stated, if any of A_i are disjunctions, then the result will not be a flat disjunction—it will have a hierarchical structure. The presence of inference rules, such as the imp2disj (see Appendix B) rule, makes hierarchical disjunctions necessary, yet there are occasions where it may be desirable for the resulting disjunction to remain flat. Of course, from several disjunctions $D_0, D_1, \dots, D_m$, using case analysis one can prove a single flat disjunction D_f containing all (or some subsets) of the formulas in each of the D_i in some order; therefore the flattening operation is a tactic for disjunctions. If all of D_i are not disjunctions, or if some of the disjunctions of D_i are to be flattened, then a single tactic may not be used and either several tactics must be created or else several proof steps must be used. Another consideration is that the tactic using case analysis would result in a realization for the disjunction which stores exactly one of the disjuncts' realizations.

Instead, we might introduce another disjunction introduction which flattens one of its disjunctions:

$$\vee\text{-intro}_h \frac{\Gamma \vdash d \rho \vee \langle A_0, A_1, \dots, A_l \rangle \quad \Gamma \vdash a_{l+1} \rho A_{l+1} \quad \dots \quad \Gamma \vdash a_n \rho A_n}{\Gamma \vdash r \rho \vee \langle B_0, B_1, \dots, B_m \rangle}$$

for $l > 0, n \geq l, m > 1$ where there exists a total 1-1 mapping $P : \{0, 1, \dots, n\} \mapsto \{0, 1, \dots, m\}$ such that for all $i \in \{0, 1, \dots, n\}$, $A_i = B_{P(i)}$. Then, for similar reasons as in the $\vee\text{-intro}_f$ definition above, we define

$$r \equiv (\text{PROJECT } p \text{ (LIST } a_0 \ a_1 \ \dots \ a_n \ \#f))$$

where $p \equiv \langle p_0, p_1, \dots, p_n \rangle$ such that each $p_i = P(i)$ maps a_{p_i} onto position i iff $a_{P(i)} \rho A_{P(i)}$ in the premises. Although this rule provides a flattening operation which remembers all of the realizations for the disjuncts of the flattened disjunction, it cannot handle all cases, and we note that more work can be done on providing useful disjunction introduction rules.

Experience so far indicates that the flattened disjunction is the more convenient statement of the disjunction. Thus for convenience, in the physical system we incorporate both $\vee\text{-intro}_f$ and $\vee\text{-intro}_h$ into the single inference scheme **disjintro** which examines the leftmost premise and uses the $\vee\text{-intro}_f$ rule if it is not a disjunction, otherwise it uses $\vee\text{-intro}_h$.

3.4.4 Case Selection

The process of case analysis can reasonably be viewed as one which chooses a proof of a theorem from the several choices presented by a disjunction. The disjunction introduction which was defined in the previous section allows us to create a disjunction with more than one possible disjunct being true. Let us formulate case analysis in a standard way to examine how non-disjoint disjunction affects case analysis:

$$\text{two-case} \frac{\Gamma \vdash d \rho A_0 \vee A_1 \quad \Gamma \vdash a_0 \rho A_0 \rightarrow B \quad \Gamma \vdash a_1 \rho A_1 \rightarrow B}{\Gamma \vdash b \rho B}$$

What is b ? To answer that first let us assume that we intend to realize disjunctions in the manner described in section 3.4.2. If $A_0 \vee A_1$ is a dichotomy, then d will always be either $\langle \#f, x \rangle$ when A_1 is true, or $\langle y, \#f \rangle$ where A_0 is true, for some expressions x and y . In either case, to get a b realizing B we can apply y to a_0 if $d = \langle y, \#f \rangle$ or else x to a_1 if $d = \langle \#f, x \rangle$. With a dichotomy the decision of which implication to use is indicated unambiguously by the proof of $A_0 \vee A_1$. But there is no reason provided by the logic as to why we should consider $A_0 \vee A_1$ to be disjoint cases (other than by biased assumptions about how disjunctions are created). Further, the analysis of cases does not specify which of $A_0 \rightarrow B$ or $A_1 \rightarrow B$ to use to conclude B when both A_0 and A_1 are true. In particular, there are two proofs of B available through two different implication elimination inferences.

Let us expand the *two-case* rule into the **caseanal** rule MIZAR-C defines:

$$\text{caseanal} \frac{\Gamma \vdash d \rho \vee \langle A_0, A_1, \dots, A_m \rangle \quad \Gamma \vdash a_0 \rho A_0 \rightarrow B \quad \dots \quad \Gamma \vdash a_m \rho A_m \rightarrow B}{\Gamma \vdash p \rho B}$$

for $m > 0$, where $d \equiv \langle d_0, d_1, \dots, d_m \rangle$. The realization paired with B must be p such that p is an expression which

1. chooses a c so that A_c is true, and $a_c \rho A_c \rightarrow B$ and $d_c \rho A_c$.
2. applies the chosen a_c to d_c to obtain the x such that $x \rho B$

If we assume that $d \rho \vee \langle A_0, A_1, \dots, A_m \rangle$ as in the interpretation in 3.4.2, properties 3.4.2.a and 3.4.2.b hold for d . Then an example of such an expression p is the one which tests each of the d_i in turn to see if it evaluates to a non-**#f** value and chooses the first c such d_c evaluates to a non-**#f** value. Call this program *First-Case*.

Clearly using *First-Case* for p preserves the soundness of the interpretation because, by property 3.4.2.b, *First-Case* will not pick a c such that A_c is false, and by the interpretation of 3.4.2, at least one d_i will be non-**#f** so *First-Case* succeeds. **caseanal** is currently implemented to perform a slight variation on the *First-Case* selection strategy and p is defined to be $p \equiv (\text{CASE-ANAL } d \text{ (LIST } a_0 \ a_1 \ \dots \ a_m))$, where **CASE-ANAL** is a program which goes through each d_i and uses the smallest k where d_k is non-**#f** for its choice of c . This is an *eager* evaluation which is significantly different than the *lazy* evaluation which *First-Case* defines. We note, however, that it is only the realizations of the disjuncts which get evaluated eagerly—the application of the realizations, $(d_c \ a_c)$, occurs exactly once. If we could ensure lazy evaluation for the list expressions for d and $(\text{LIST } a_0 \ a_1 \ \dots \ a_m)$, then we could expand the **CASE-ANAL** expression into the Scheme special form **CASE** with the d_i in place as the case tests and $(a_i \ d_i)$ as the expression to evaluate for those tests which evaluate to non-**#f** values.

3.4.5 Conjunction Inferences

The conjunction $\wedge \langle A_0, A_1, \dots, A_m \rangle$ intuitively means that all the formulas A_i are simultaneously true. The calculus of conjunction manipulation allows for the rearrangement of the A_i , the addition of any number of other proven formulas, and the subtraction of any of the A_i . Each of these operations can be, and often are, written as separate rules of inference which are used in combination. Nevertheless, because of our definition of conjunction, we may reasonably treat all conjunction inferences within one inference scheme, making conjunction manipulation uniformly simple. The treatment of conjunction inferences can be made uniform by writing a conjunction inference as the choice of a subset of formulas from the sequence of formulas.

The *conjunction* scheme is

$$\text{conjunction} \frac{\Gamma \vdash c \rho \wedge \langle A_0, A_1, \dots, A_n \rangle \quad \Gamma \vdash a_{n+1} \rho A_{n+1} \quad \dots \quad \Gamma \vdash a_m \rho A_m}{\Gamma \vdash b \rho B}$$

for $n > 0$ and $m \geq n$ where all of $A_0, A_1, \dots, A_n$ are not conjunctions but each of $A_{n+1}, A_{n+2}, \dots, A_m$ may be (flat) conjunctions, and B may be a conjunction. Now

consider the formula sequence C_m where $C_i, i \geq n$ are the sequences defined as:

$$C_n = A_0, A_1, \dots, A_n$$

$$C_i = C_{i-1}, \begin{cases} A_i & \text{if } A_i \text{ is not a conjunction} \\ A_{i0}, A_{i1}, \dots, A_{ik} & \text{if } A_i = \bigwedge \langle A_{i0}, A_{i1}, \dots, A_{ik} \rangle \text{ for some } k > 0 \end{cases}$$

Then B may either be a conjunction $\bigwedge \langle B_0, B_1, \dots, B_l \rangle, l > 0$ such that for all $i \in \{0, 1, \dots, l\}$, $B_i = A_k$ for some k where A_k is in C_m , or B can be any single formula from C_m . Our implementation of *conj* creates a program for b which projects or selects the appropriate realizations from the premises. When B is a conjunction,

$$b \equiv (\text{PROJECT } (\text{LIST } p_0 p_1 \dots p_l) (\text{LIST } c_n c_{n+1} \dots c_m))$$

where $c_i \rho C_i$ and $A_{p_i} = B_i$ for all $i \in \{0, 1, \dots, l\}$. If B is a single formula,

$$b \equiv (\text{LIST-REF } (\text{LIST } c_n c_{n+1} \dots c_m) k)$$

for the above k . For clarity at the proof level, we say that $\bigwedge \langle A \rangle$ for the single formula A does not define a conjunction. Nevertheless, if we look at the formula A as the "conjunction" of the single formula $\bigwedge \langle A \rangle$, then the *conjunction* rule becomes uniform: it creates a single conjunction from a list of them, we simply interpret conjunctions of the form $\bigwedge \langle A \rangle$ as we would the formula A .

As a final note about the *conjunction* inference rule, as might be expected, we could have defined it to take both conjunctions and non-conjunctions for every premise, not merely the leftmost one. Firstly though, *conjunction* provides the generality we need for the rule since it implements both of the standard conjunction introduction and elimination rules. Secondly, the statement of *conjunction* would have been made unnecessarily complicated, since as it is it is clearly generalizable. MIZAR-C in fact implements the *conj* scheme as this more general form of *conjunction*, which recognizes conjuncts in any or all positions in the premises and flattens them appropriately.

3.5 Logic with Equality

Equality is normally viewed as a predicate with special status and it is this status which introduces a calculus on terms in the logic: a calculus of term replacement. Potentially, one could introduce equality into the logic as such a distinguished predicate and some second-order axioms about substitutions based on equality. Because MIZAR-C does not have the capability of stating or using second order assumptions, our equality reasoning is formulated instead through a distinguished predicate with special syntax and a set of inference schemes, *take*, *eqintro*, and *equality* which supply the necessary machinery to perform the equality substitutions.

Through these inference rules, MIZAR-C provides an equality notation and calculus for terms which we interpret at a very low level, and because of this, we do not

yet discuss the problems involved with extensionality versus intentionality. When we write $s = t$ for arbitrary terms s and t , we mean that s is realized by an expression which is *computationally equal* to that which realizes t . Another meaning we can use for this equality is that, whatever models we are using to represent Scheme expressions using terms, whenever we can assert $s = t$, the realizations associated with s and t can be exchanged in any formula F 's corresponding realization f without affecting the soundness of F 's interpretation. This idea can be made slightly more precise by saying that $s = t$ implies that *in the current context*, $\nu(s)$ can be substituted for $\nu(t)$ in arbitrary expressions x involving $\nu(t)$, where $x \rho X$ for some formula X , so that the new expression also realizes X .

3.5.1 Equality Introduction Rule

MIZAR-C's proof checker keeps track of which variables are defined and which are not, so is possible to check if a variable denotes a value, or simply *denotes*. If a variable denotes, we are permitted to infer that it is equal to itself (reflexivity). This rule which can be written as:

$$\text{eqintro} \frac{\Gamma^v \vdash \Delta}{\Gamma^v \vdash \Delta, \#t \rho v = v} .$$

Here $v = v$ is the special syntax for the binary equality predicate, and Γ^v indicates that v is some variable defined in Γ (unless v was introduced using the **consider** mechanism from an existentially quantified formula marked non-constructive). The first thing to notice about the inference is that we say the Scheme language constant **#t** realizes $v = v$ for any v . In fact, we do not interpret any equality operations as computational operations, so in truth there is no realization associated with an equality formula. Nevertheless, as we shall see with the **take** and **equality** schemes, we need to remember if the equality was obtained in a context where the symbols of the terms in the equality have constructive meaning. Thus we use **#t** as an indicator that the v in $v = v$ has constructive meaning for the benefit of the other inference rules such as existential introduction.

3.5.2 Take Rule

The language of the logic does not have a mechanism for explicitly introducing an arbitrary function term. The only way to write a complex term in any context is to write it as part of a formula which was restated from the environment or was obtained through a valid deduction from assumptions in the environment (including an **assume** statement, or through the Axiom of Choice if the system is extended with it). However, in section 3.3.1 we have already introduced the notion of fully defined formulas and the set constructor $\mathcal{D}$ which gives the set of complex terms which are asserted for any formula stated in the current context. From $\mathcal{D}$'s definition, we can

define a rule which extracts a term in $\mathcal{D}(A)$ for any formula A :

$$\text{take} \frac{\Gamma \vdash a \rho A}{\Gamma \vdash a \rho A, \#t \rho t = t}$$

where $t \in \mathcal{D}(A)$. Clearly $\mathcal{D}(t = t) = \{t\}$ if $t \in \mathcal{D}(A)$, and substituting $\nu(t)$ for $\nu(t)$ in any Scheme expression x leaves x unchanged so substitutions on any formula F using this equality preserves the soundness of the interpretation of equality.

3.5.3 Substitution Scheme

Given an equality $a = b$ and a formula A in term a , we may substitute b for any a in A . In addition to the reflexivity property which **take** and **eqintro** embody, equality has two other properties: transitivity and symmetry. To make the substitution scheme useful, it should encompass both these properties, so we define it in the following manner.

$$\text{equality} \frac{\Gamma \vdash x \rho A \quad \Gamma \vdash \#t \rho a_0 = a_1 \quad \dots \quad \Gamma \vdash \#t \rho a_{2m} = a_{2m+1}}{\Gamma \vdash x \rho B} \quad m \geq 0$$

where A and B are *equality unifiable* under the set of equalities given. For equality unifiable formulas A and B , x ($\#f$) realizes B as it did A (see [23] for a similar interpretation), indicating that we have dealt only with notational differences.

The **equality** scheme implements two different operations for convenience since although they can be performed simultaneously, merging their operations requires a more involved description of the process. Hence we have split the inference into the cases where $m > 0$ and where $m = 0$ and perform different operations for each. For the case where $m > 0$, in three steps we define equality unifiability as:

1. Let E be the equalities given in the inference, $a_0 = a_1, \dots, a_{2m} = a_{2m+1}$. From E form equivalence classes C_j where $a_x, a_y \in C_k$ iff $a_x = a_y \in E$. Then pick a single member from each C_j , c_j called a *representative* of that equivalence class.
2. For each occurrence of a term a_j in A and B , replace (substitute) its occurrence with the representative, c_k where $a_j \in C_k$, to form the unifiers A' and B' .
3. Then A and B are equality unifiable under E iff $A' = B'$.

When $m > 0$, **equality** implements reasonings based on the transitivity of equality by virtue of the construction of the equivalence classes C_i .

For the case where $m = 0$, no equalities are given for substitution, so A and B are said to be equality unifiable if they are equal under the symmetric nature of equality. More specifically, A and B are equality unifiable for $m = 0$ if $a_0 = b_0, a_1 = b_1, \dots, a_j = b_j, j \geq 0$ are equality subformulas in A , and A can be transformed into B by substituting $b_i = a_i$ for some or all of $a_i = b_i$.

The soundness of the interpretation of the inferences allowed by the equality scheme depends on whether or not x realizes A' as well as A . If $m = 0$ then the realizations of A and B are identical since all equalities are realized by $\#t$. Otherwise, for $m > 0$, if $\nu(a_{2i})$ can be substituted for $\nu(a_{2i+1})$ for any of the equalities in E , then any $\nu(a_x)$ can be substituted for $\nu(a_y)$ if $a_x, a_y \in C_j$ for some j . The substitution is allowed because we insist that two terms a and b can be used to state $a = b$ in the context only if $\nu(a)$ and $\nu(b)$ are computationally equal. Therefore x realizes A' if it realizes A , and similarly it realizes B if it realizes A' and $A' = B'$.

To complete our discussion of the equality calculus, we must examine the other inference rules to ensure that they preserve the property that equalities indicate substitutability on the logical level and on the computational level. Only the quantifier rules are permitted to change a term in an equality. Since all the quantifier rules are interpreted so that substitutions on the logic level correspond exactly to substitutions at the realization level, the other inference rules preserve the equality rules' soundness.

3.6 Sorted Variables

Each logic variable is attributed with a *sort* which is indicated by a single identifier. We interpret sorts as dividing the universe of discourse into distinct sets such that a variable can name an object which is a member of exactly one sort, the sort which the variable was declared to be quantifying over. All of the inference rules in the previous sections were written with the assumption that the sorts of all variables were identical, and the syntax of the language allows for the user to leave out explicit specification of the sort of a variable. If the sort designator for a variable is omitted, its sort is defaulted to the special sort **thing** (unless the default sort for the identifier is changed using the special statement **reserve** in the environment for the proof). If a user wishes to completely remove sorts from consideration, it is entirely possible by simply omitting them everywhere and the inference rules, which we have defined without taking into account the sorts, work as stated: the system using a single sort is trivially isomorphic to an unsorted one. The following constraints modify the applicability of the inference rules when using variables with heterogeneous sorts:

1. We define the sort of a complex term $t = (f\ t_0\ t_1\ \dots\ t_n)$ to be the sort of the function variable f in t . This feature makes it impossible for sorts to represent types, but it simplifies term formation and ensures that variables which name terms (such as during value capture) have matching sorts.
2. Sort is significant for quantification. The sort of the quantified variable being introduced must match the sort of the term being quantified over, and the sort of a quantified variable being eliminated must match the sort of the variable eliminating the quantifier. Also, if formula A equals B without considering the sort constraints, then the sorts of all bound variables in a formula A must also

match the sorts of their respective bound variables in B for A to be considered equal to B .

Currently, the tagging of variables and terms with sorts in no way changes the interpretation of the logical inferences since the rules make no reference to them. However, sorts are significant for the realization of axioms since they alter the applicability of inferences on the variables and terms in them. The constraints which differing variable sorts place on the applicability of inference rules can be employed in formulating axioms to indicate that the object universe is separated into disjoint universes identified by sort names. Disjoint sort universes might be utilized, for example, to separate objects of the concrete computation universe from meta-level objects such as propositions or formulas.

Chapter 4

Frameworks for Relating Programs to Specifications

Chapter 3 introduced the interpretation of the logic calculus as Scheme operations, which we use to implement an extraction mechanism. However, we did not explicitly refer to the realizations paired with axioms or assumptions in hypothetical reasonings. In order to retain the soundness of our interpretation of the logical operations as Scheme operations on assumptions in reasonings, we must make certain *requirements* of the Scheme expressions which are paired with them. For the same reason, these requirements are also imposed on all added axioms in order to ensure the soundness of the extended system. These requirements can be made explicit using the notion of the *shape* of a formula and realization. In the following sections we introduce the definition of shape and discuss schemes for pairing axiom formulas to realizations so that we ensure the programs extracted from proofs from these axioms satisfy their specification.

The interpretation we have defined in Chapter 3 is only one of many possible interpretations which result in a useful realizability interpretation. In particular, the interpretation was made such that if the axioms are purely functional, then all extracted programs will be functional also. We will show that changes may be made to the extraction machinery to accommodate the extraction of programs which are correct with respect to their specification, but which are no longer purely functional. Specifically, we show that, with particular parallel execution machinery available, case analysis can be made to exploit the parallel machinery in order to choose efficient computations from a selection of them. We also show that existential elimination is formulated in our system to take advantage of nondeterministic case analysis, as well as afunctional realizations which are paired with axioms.

Together, the scheme of pairing formulas with axioms and the assignment of an interpretation to the logical calculus outline a rough *framework* for encoding both purely functional and not purely functional program-specification relationships, outlined in Chapter 1.

4.0.1 Shape

Conceptually, the realizations paired with assumptions or axioms are imagined to be indistinguishable from those extracted from correct constructive proofs using our interpretation. Conversely, the realizations extracted when interpreting a proof define a *shape* for the realizations which corresponds to the shape of the formula they are paired with. It is a general dogma of the interpretation that the shape of a formula specifies the shape of its realization.

The correspondence between the shapes of the formulas and the realizations paired with them, can loosely be written as a macro $\mathcal{S}$ in a set of recurrences:

$$\begin{aligned}
 \mathcal{S}(A) &\equiv \#t \text{ for atomic } A: \text{ the predicates, } \perp, \text{ and } = \\
 \mathcal{S}(\neg A) &\equiv \#t \\
 \mathcal{S}(\bigvee \langle A_0, A_1, \dots, A_m \rangle), m > 0 &\equiv (\text{LIST } \mathcal{S}(A_0) \mathcal{S}(A_1) \dots \mathcal{S}(A_m)) \\
 \mathcal{S}(\bigwedge \langle A_0, A_1, \dots, A_m \rangle), m > 0 &\equiv (\text{LIST } \mathcal{S}(A_0) \mathcal{S}(A_1) \dots \mathcal{S}(A_m)) \\
 \mathcal{S}(A \rightarrow B) &\equiv (\text{LAMBDA } (v) \mathcal{S}(B)) \\
 &\quad \text{for some Scheme variable } v \text{ not occurring} \\
 &\quad \text{bound in } \mathcal{S}(B) \\
 \mathcal{S}(\forall x. A) &\equiv (\text{LAMBDA } (v) \mathcal{S}(A)) \\
 &\quad \text{for some Scheme variable } v \text{ not occurring} \\
 &\quad \text{bound in } \mathcal{S}(A) \\
 \mathcal{S}(\exists x. A) &\equiv (\text{LIST } e \mathcal{S}(A)) \\
 &\quad \text{for some Scheme expression } e
 \end{aligned}$$

Negated formulas are not interpreted as containing any constructions since none of our constructive inference rules are capable of producing a negated formula $\neg A$ where it did not exist already. In those cases where the formula was deduced through non-constructive inference rules, the formula will be marked non-constructive and will be realized with $\#f$. Nevertheless, negated formulas can be useful for writing constructive proofs since they can provide the knowledge of when a proof of a formula is impossible. For example, if we have defined $\text{Nat}[x]$ to mean x is a natural number, and succ is the successor function, we can write our knowledge that 0 is not the successor for any natural number as:

not0Succ: for x holds $\text{Nat}[x]$ implies not $(\text{succ } x)=0$

which can be read as a statement that the formula $(\text{succ } x)=0$ for x satisfying $\text{Nat}[x]$ cannot be constructively proven. It is safe to mark an axiom such as **not0Succ** as constructive because no term can be extracted from any subproof using the axiom without the term already being defined elsewhere. To see this notice that $\mathcal{D}(\forall x. \text{Nat}[x] \rightarrow \neg(\text{succ } x) = 0) = \emptyset$, $\mathcal{D}(\text{Nat}[x] \rightarrow \neg(\text{succ } x) = 0) = \emptyset$ for any x , and $\mathcal{D}(\neg(\text{succ } x) = 0) = \emptyset$ for any x satisfying $\text{Nat}[x]$, by the definition of $\mathcal{D}$. Therefore, for any use of the formula **not0Succ**, neither **take** nor **exintro** may be used to extract a term from **not0Succ** or its descendents even if it is marked constructive. We currently include **negintro** in the list of non-constructive inference rules

```

NatDef: ex 0 st Nat[0]
      & (ex succ st
        (for i st Nat[i] holds Nat[(succ i)] & not (0 = (succ i)))
        & (for i,j st Nat[i] & Nat[j] holds
          (not ((succ i) = (succ j)) implies not (i=j))
          & ((succ i) = (succ j) implies i=j)
        )
      )
rby nat-def;

```

Figure 4.0.1: A representation of a data type for the naturals.

in Appendix B, but for the same reasons as for using negated formulas in axioms, negation introduction may also be marked constructive.

4.0.2 Adding Common Structures

Predicates are often realized simply by `#t` since they are usually interpreted as being atomic expressions which indicate membership in a set of values—those values of its arguments for which the predicate is true. Predicates can be used in an axiomitization to indicate sets of values or *data types* by introducing them in a single existential formula which acts as a *package* for the data type. Figure 4.0.1 gives an illustration of an existential formula which is an effective definition of the data type for naturals.

The `rby` mechanism employed in Figure 4.0.1 pairs a Scheme sentence named `nat-def` with the formula at label `NatDef` and `nat-def` is defined as a Scheme variable which evaluates to the expression realizing the formula at `NatDef`. The sentences which the variables `0` and `succ` quantify over define how those values can and cannot be used, and as such are often called the *module* definition. The use of `Nat` in the universal sentence “for `i` st `Nat[i]` holds F ” is intended to be a restriction on variable `i` for formula F , and is commonly interpreted to mean $\forall i \in Nat. F$.

It is usually an assumption (tacit or explicit) in such definitions that the set defined by `Nat` is the smallest set satisfying the axioms which are encoded in the package. In our case, it is an explicit assumption that the definition at `NatDef` is the only axiom in the axiom set which allows the conclusion of a formula satisfying the predicate `Nat`. With that assumption then the set of theorems constructively provable from the assumption of `NatDef`, using the inference rules available in the extension, defines the set of values which establish `Nat`. Then `NatDef` is a sound encoding of how we conceive (and implement in `nat-def`) the set of naturals if the provable formulas establish `Nat` only on those terms in `succ` and `0` which are interpreted as Scheme expressions which encode a natural number. If `succ` is mapped to `1+` and `0` to `0` then by induction on the length of derivation using the base inference rules we can be convinced that `NatDef` allows proofs of theorems which are sound for the arithmetic

```

consider 0 such that 1: Nat[0] &
  (ex succ st
    (for i st Nat[i] holds Nat[(succ i)] & not (0 = (succ i)))
    & (for i,j st Nat[i] & Nat[j] holds
      (not ((succ i) = (succ j)) implies not (i=j))
      & ((succ i) = (succ j) implies i=j) ))
  by direct(NatDef);
ZeroNat: Nat[0] by conj(_PREVIOUS);
consider succ such that
  2: (for i st Nat[i] holds Nat[(succ i)] & not (0 = (succ i)))
    & (for i,j st Nat[i] & Nat[j] holds
      (not ((succ i) = (succ j)) implies not (i=j))
      & ((succ i) = (succ j) implies i=j) )
  by conj(1);
...

```

Figure 4.0.2: Proving lemmas from a compact representation of a data type.

model for $1+$ and 0 .

NatDef is a closed formula which introduces our notation for the naturals. When a data type is introduced through a single existential statement such as **NatDef**, we call the axiom a *compact representation* of the data type. The proof writer must use the **consider** statement to obtain the values of 0 and the successor in order to use this definition; the axiom as it is stated is not easy to use in a proof. One can quite easily rearrange **NatDef** at the top level of the proof text (not within a **now** reasoning) into more convenient lemmas, an example of which are given in Figure 4.0.2. The proof introduces the two constants 0 and **succ** to the scope, and as demonstrated for **ZeroNat**, the remainder of Peano's axioms can be provided using short proofs from 2.

It is likely that the proof writer will require the steps at 1 and 2 and will desire the statement of the Peano axioms in a more convenient form. Therefore it may make sense to create the extension defining the data type originally in the *diffuse* format of lemmas which may be proven from the compact representation. Of course with a tactic mechanism, it would be possible to provide tactics along with the extension which extract each of the required lemmas. As an alternative, MIZAR-C provides **import** and **export** mechanisms which allow the lemmas proven in one proof text to be imported as axioms in another proof. If the extension writer is uncertain about writing an extension in the diffuse format, the extension can be written in the compact form and a set of usable lemmas proven from it can be saved into an importable text using the **export** mechanism. Then the exported lemmas may be loaded into the proof which requires the lemmas as axioms using **import** facility.

Even though we have declared **NatDef** to be a definition for a data type for natural

number representation, no explicit *definition* of the predicate **Nat** was made so that the system could check for improper, or multiple, definition. The extension writer has been held entirely responsible for writing the axioms in such a way that the axioms, along with the inference rules, result in a sound system. Although this might be acceptable for the axiomatization of basic data types and operations, the proof writer using these basic definitions is also responsible for defining any new properties or operations for the basic definitions. For example, in order to prove the existence of the summation function $\sum_{i=0}^z f(i)$ for arguments $z \in \text{Nat}$ and $f : \text{Nat} \mapsto \text{Nat}$, the proof writer must be able to define a predicate P (or formula F) in f , z , and some variable sum for which P (or F) is true only for a single value of sum for set f and z . We show how we currently handle these definitions in section 5.1 noting that it is a widely studied problem (see for example [9, 15, 13]).

4.0.3 Afunctional Realization Specification

As mentioned in section 3.3.2, the intuitive understanding of the logical formula $\exists x.P$ for P with x in it, is that $\exists x.P$ specifies that some elements from a nonempty set of values from the universe of discourse satisfying P . When we wish to claim that $\exists x.P$ specifies that exactly one value satisfies P (as we wish for indicating functions), we normally require the assumption of, or proof of, $\exists x.P \wedge \forall y.(P_{x \leftarrow y} \rightarrow y = x)$. If we cannot prove $\exists x.P \wedge \forall y.(P_{x \leftarrow y} \rightarrow y = x)$, only $\exists x.P$, then for certain P we are free to interpret P as meaning there are multiple choices for x which satisfy P . In other words, we can interpret $\exists x.P$ as a set if P contains no free variables, or we can interpret it as a relation if it does. As a trivial example of a P which we are not free to interpret as a relation, if we assume $\exists x.x = z$ holds for some free variable z , then the formula $\forall y.y = z \rightarrow y = x$ may be derived by use of the transitivity of the equality predicate. In this case, we are forced to interpret $\exists x.x = z$ as a function since we can prove it behaves like one. The subformula $\forall y.(P_{x \leftarrow y} \rightarrow y = x)$ in $\exists x.P \wedge \forall y.(P_{x \leftarrow y} \rightarrow y = x)$ acts as a *certificate* of the functional behaviour of the formula P .

$\exists x.P$ is forced to be functional only if it contains an equality which relates bound variables to free variables, as above, because we have fixed the meaning of the equality predicate to be functional. In the formula $\exists x.P$, if x is not related to any free variables by the equality predicate, this restriction is not imposed by MIZAR-C and it is possible to interpret $R \equiv \exists x.P$ as specifying any value from a set of values whose cardinality is greater than one. It would be impossible, in general, to realize $R \equiv \exists x.P$ such that each member of this set is recorded in the realization since, for example, R might specify an infinite set. A more practical solution is to realize R by a procedure which chooses a value from the set of possible values whenever it is executed. In that way, when its value is considered using the **consider** statement, the program is executed and its value is bound to the value returned by the procedure.

As an illustration, for the sentence $\exists x.Q$, assume the axioms in use do not give us knowledge about Q which forces us to interpret Q as a function, and therefore we

are free to choose x which satisfies Q from a set. Then we can implement $\exists x.Q$ as the pair constructor (LIST p s) with the following restrictions:

1. p is an expression (procedure) which returns any one of a set of values when evaluated.
2. for any value v which p returns, it must be equal to the value of
 - $\nu(x)$ for a logical variable identifier which establishes Q
 - $\nu(f\ t_0\ t_1\ \dots\ t_n), n > 0$ for some term $t = (f\ t_0\ t_1\ \dots\ t_n)$ such that t is in $\mathcal{D}(Q_{x \leftarrow (f\ t_0\ t_1\ \dots\ t_n)})$ and $Q_{x \leftarrow (f\ t_0\ t_1\ \dots\ t_n)}$ is true

An example of a formula which might usefully exploit the possible ambiguity in the choice which is made at a **consider** statement using $\exists x.Q$ is the axiomatization of a pseudo-random number generator. Let $Q \equiv \text{Nat}[x]$ above, and let

$$(\text{LIST } (\text{RAND})\ \#t)\ \rho\ \exists x.\text{Nat}[x]$$

where $\text{Nat}[x]$ is interpreted as meaning x is a natural number and where **RAND** is a procedure which implements some pseudo-random number generator which returns natural numbers as values. Then each choice of x satisfying $\text{Nat}[x]$ above which was introduced through the use of the **consider** statement may be different since each choice executes **RAND** and binds the value it returns to $\mu(x)$ (so long as the list construction is lazy, see Chapter 5 for more). As required, if we cannot show that $\text{Nat}[x]$ must be a single value (and we should not be able to if **Nat** represents the natural numbers data type) we will not be able to show that two variable x and x' **consider**'d from the above existentially quantified formula are equal. If $\text{Nat}[x]$ is true in the current extension for any natural number, then the above realization is correct with respect to the specification $\exists x.\text{Nat}[x]$.

As can be expected, the extension implementor must be careful in making the choice of allowing existentially quantified formulas to be realized by expressions which behave afunctionally since it can affect the soundness of the available inference rules. For example, if an extension to the natural numbers is added which admits an afunctional realization, yet the use of the Axiom of Choice rule for natural numbers is desirable, the statement of the choice rule must be made so it does not attempt to extract a function where only a relation exists. To perform this check and be considered safe in the presence of afunctional realizations for existentials, the rule must insist that a certificate of functional behaviour be established. An example of such a rule of unique choice can be given as:

$$\text{choice}_{N\text{-safe}} \frac{\Gamma \vdash u\ \rho\ \forall x.(\exists y.P) \quad \Gamma \vdash c\ \rho\ \forall x,y,z.(P \wedge P_{y \leftarrow z}) \rightarrow y = z}{\Gamma \vdash g\ \rho\ \exists f.(\forall x.P_{y \leftarrow (f\ x)})}$$

for any formula P in x . We will not concern ourselves with the realizations u , g , and c since they are not important to the following discussion. The sentence $C_P \equiv$

$\forall x, y, z. (P \wedge P_{y \leftarrow z}) \rightarrow y = z$ acts as a certificate of the functional relationship between y and x in the formula $\forall x. \exists y. P$, if it can be proven then y must be functionally related to x . C_P will not be provable and the choice rule not usable if the environment did not provide enough information to conclude C_P .

Claim 4.0.3 Although *choice_N-safe* is written with C_P marked constructive, a non-constructive proof of C_P will also ensure the safety of *choice_N-safe*.

Proof A (classical) argument for why a classical proof of C_P suffices is as follows. Either the relation $R[x, y] \equiv \forall x. \exists y. P$ is not a function, or it is. If it is not a function, then C_P is not provable, even classically, or the context contains a contradiction. If $R[x, y]$ is a function, then even when a constructive proof of it is given, the construction from the proof is not required for the construction of g , so a classical proof is acceptable because we do not need the witness.

An illustration may help: there are many ways to sort a finite sequence of naturals into non-decreasing order, yet all such sort procedures are *functions* from sequences to sequences because the resulting list is unique. The proof of uniqueness of the sequence ordered by non-decreasing elements is a result which could be proven classically and is independent from the implementation of the sort procedure: in particular it is a kind of non-computational precondition to the correct application of the choice rule. In the case of the sorting function, it might also be reasonable to view the proof of functionality as a non-computational postcondition of the sort function.

Here then is an instance where the classical reasonings allowed by MIZAR-C can be used to advantage if *choice_N-safe* can be redefined so that its C_P premise is marked non-constructive. Allowing classical reasoning for the functionality certificates can often lead to much shorter proofs of the functionality than using only purely constructive reasoning.

Only for the case where we have made sure that all realizations corresponding to proven formulas behave functionally are we able to interpret the rule:

$$\text{choice}_N\text{-unsafe} \frac{\Gamma \vdash u\rho \forall x. (\exists y. P)}{\Gamma \vdash g\rho \exists f. (\forall x. P_{y \leftarrow (f\ x)})}$$

choice_N-unsafe will not be sound in the presence of afunctional axioms since the function f may then not exist. In addition, in the next section we show that it is possible to extract realizations from proofs which are afunctional, thus there are cases where the choice rule must insist on a certificate of functionality, even if no axioms are paired with afunctional realizations.

4.0.4 Alternative Case Selection Strategies

In our framework for relating theorems to programs (see chapters 1 and 4), theorems are viewed as specifications of the desired relation between input and output. Since

there can be more than one proof of a specification, there may be more than one non-isomorphic program which can be proven to realize the specification. Changing the proof of a theorem is not significant to the theorem's logical statement, since any valid proof verifies the theorem. However, in terms of the running times of extracted programs, it is possible to make a potentially significant difference by changing the proof of the theorem.

Our hope here is to show that more imaginative interpretations of case analysis may improve the computational complexity involved in case selection. Recall from section 3.4.4 that for the rule

$$\text{caseanal} \frac{\Gamma \vdash d \rho \vee \langle A_0, A_1, \dots, A_m \rangle \quad \Gamma \vdash a_0 \rho A_0 \rightarrow B \quad \dots \quad \Gamma \vdash a_m \rho A_m \rightarrow B}{\Gamma \vdash p \rho B}$$

for $m > 0$, where $d \equiv \langle d_0, d_1, \dots, d_m \rangle$, the current case selection strategy for p is *First-Case*. For computational complexity reasons, we may wish to stipulate that p must choose a $c \in \{0, 1, \dots, m\}$ so that the pair a_c, d_c , where $(a_c \ d_c) \rho B$, executes in the fewest computation steps.

Of course, in general p cannot always perform this choice by merely inspecting the a_i and d_i . Nevertheless, with suitable knowledge of the realizability interpretation built into the program extraction machinery, it may be possible to obtain a rough or possibly exact count of the number of operations each a_i will incur as a result of executing on d_i . If this accounting is available, we could hope to decide at proof time which proof ($\rightarrow$ -elimination) to use to establish C and write the desired p from this decision. Another possible selection strategy is a permuted-order testing which may be useful if something is known about the relative frequency with which each case occurs so that the entropy of the selection is maximized.

However, even without operation counting in the extraction procedure, with the appropriate computation engine, a p could indeed be built which decides which a_i, d_i pair executes in the fewest steps. For example, if p is written to be executed on a parallel machine with a *eureka jump* mechanism, it can run each a_i, d_i pair, where d_i is non-#f, on separate machines and then (retroactively) choose the result of that a_i, d_i pair which finishes its calculations first.

As an aside, an unexplored consequence of the above observation is that, with parallel execution at our disposal, as long as one of the $(a_i \ d_i)$ applications specifies a halting computation any (or all) of the other $(a_i \ d_i)$ need not indicate halting computations. In particular, as long as $(a_h \ d_h)$ for some h halts (and of course d_i is non-#f) any of the $a_i, i \neq h$ may be an improper extraction from a proof of $A_i \rightarrow B$ with a non-constructive step in it, provided that $(a_i \ d_i)$ does not halt or halts with an error condition so p does not choose it.

Finally, observe that the realizations extracted from the **consider** statement are correct even in the case of afunctional choice since the expression (CAR e) (in the statement of the inference) is evaluated exactly once to obtain its value and which is then saved by the (LET ...) construction which binds that chosen value to a variable. Figure 4.0.3 gives a small incomplete example where we use two proofs of the

```

now
  let x; assume d: A[x];
  case1: now assume A[x];
    .....
    thus (ex y st B[x,y]) by exintro(_PREVIOUS); end;

  case2: now assume A[x];
    .....
    thus (ex y st B[x,y]) by exintro(_PREVIOUS); end;

  A[x] or A[x] by disjintro(d,d);
  thus (ex y st B[x,y]) by caseanal(_PREVIOUS,case1,case2);
end;

for x st A[x] holds (ex y st B[x,y]) by direct(_PREVIOUS);

```

Figure 4.0.3: An example of parallel case selection.

same relation to prove a new relation whose realization may execute afunctionally if the case selection is performed nondeterministically as given above. The ellipsis in deductions at **case1** and **case2** indicate distinct deduction sequences which establish **ex y st B[x,y]** from **A[x]**. The case analysis might choose either of them depending upon which proof results in an extraction which runs quicker on the given input value **x**. Notice also that if we do not have a guarantee that **B[x,y]** functionally relates **y** to **x**, then we cannot guarantee that the value for **x** captured by the existential quantifier in **case1** and **case2** will be the same.

Chapter 5

Some Problems Encountered

MIZAR-C's implementation is not complete with respect to the descriptions of the support mechanisms in Chapters 2 and 3, such as the marking scheme. We have also found that some features such as the extraction mechanism might be reasonably extended in order to become useful for encoding a variety of program-specification relationships. We outline these problems associated with the actual implementation of MIZAR-C's proof checker and program extraction mechanism in section 5.2.

We note that, other than defining a realizability interpretation and implementing a basic method of extending the system, we have incorporated essentially no features which aid in general program development. Although we have not pursued the implementation of these features, the need for them is recognized, and in section 5.3 we discuss some problems encountered when writing the type information of objects, and when trying to define predicates. We highlight these problems and give an extended example of a proof in MIZAR-C in section 5.1.

5.1 An Example Proof

We give an example of using MIZAR-C on a problem which is admittedly small but which demonstrates a use of formulas as specifying types, and a use of extensions to the system including new inference rules. The proof is of the summation function and it appears in Appendix C.

In the proof, the formula at **SigmaDef** is the *definition* of the summation operator. Since there is no predicate definition mechanism yet for the proof writer, we are forced to include the (impredicative) definition of the predicate **Sigma** using the **magic** rule. **magic** is a rule which can be used on any formula and uses the definition of **shape** defined in section 4.0.1 to create an appropriate realization for the conclusion. We will not discuss the **magic** rule, except to say that the **magic** rule should be thought of as adding axioms to the system without being required to manually assign a realization to them. For example, the axiom

NatT-0: for **x** st **Nat[x]** holds **Nat[(succ x)]**;

may be added to any system using the **magic** rule since it assigns a realization such as:

$$(\text{LAMBDA } (x) \ (\text{LAMBDA } (\text{natx}) \ \#t) \)$$

to the formula where x is $\nu(x)$ and natx is the realizer variable for the $\text{Nat}[x]$ antecedent in NatT-0 .

The **extendwith naturals-t** statement in the environment section loads in the predefined extension which implements the basic data type for naturals, much as was done in section 4.0.2. It defines the constant 0, the function $\text{succ} : \text{Nat} \mapsto \text{Nat}$, and we interpret the predicate $\text{Nat}[t]$ for arbitrary defined term t to mean that $t \in \text{Nat}$. It also defines an induction inference rule **natind**:

$$\text{natind} \frac{\Gamma \vdash bc \rho P_{x=0} \quad \Gamma \vdash is \rho \forall x. \text{Nat}[x] \rightarrow (P \rightarrow P_{x=(\text{succ } x)})}{\Gamma \vdash p \forall x. P}$$

for any formula P in variable x , where succ is the successor function succ . p in **natind** is defined to be

$$(\text{LAMBDA } (x) \ (\text{LAMBDA } (\text{Natx}) \ (\text{NAT-IND-ITER } x \ bc \ is) \) \)$$

where x is $\nu(x)$, Natx is the realizer variable for the antecedent $\text{Nat}[x]$ in the conclusion. **NAT-IND-ITER** is a Scheme combinator defined in the extension which recursively applies is starting bc , x times and returns the result. The application is performed as

$$(\ (is \ r) \ \#t)$$

where $\#t$ represents the realization of $\text{Nat}[x]$ and r is the currently iterated realization which starts out as bc . The extension **nat-plus-t** adds to the system the **plus** function of type $\text{Nat} \times \text{Nat} \mapsto \text{Nat}$ which adds two numbers together. All axioms from either extension which are used in the proof have been restated just after **SigmaDef** using labels starting with **R** (the **direct** rule allows one to restate any formula). In case the examples before have not made it clear, the label **_PREVIOUS** in a deduction stands for the formula immediately before the current one.

5.2 Implementation Restrictions

At present, the marking strategy defined in section 3.1 is not implemented by **MIZAR-C**, which means that construction errors are not mechanically caught at proof time. Nevertheless, the non-constructive inference rules pair realizations of $\#f$ with their conclusions and $\#f$ is an invalid realization for a true formula. Because of this feature, the construction error tests may be quite easily added to the current implementation. As a convenience, the marking of each formula as constructive or non-constructive might also be indicated textually by the system so that the proof writer can immediately identify which formulas do not have a constructive proof. Without the textual marking of the formulas as constructive or non-constructive, one

cannot immediately know from a proof where non-constructive steps have prevented a correct program from being extracted.

A more serious concern is that we currently use Scheme as the execution machine for the extracted programs. Although this is convenient for system development reasons, it is poor with respect to efficiency concerns because Scheme fully evaluates the list constructor. That is, all elements x_i of $(\text{LIST } x_0 \ x_1 \ \dots \ x_m), m > 0$ are evaluated to produce $(\overline{x}_0, \overline{x}_1, \dots, \overline{x}_m)$. All of our realizations reduce to normal form such that if $x \rho Y$ then $\overline{x} \rho Y$ also because we have been careful to evaluate any afunctional expressions at most once by capturing values using the LET special form. Therefore eager evaluation does not introduce execution errors, but unnecessary evaluations usually occur as a result. Execution order is especially important during case analysis because, for example when using the *First-Case* program, not all elements in a disjunction always need be evaluated since only the first non-#f element is important. Since Scheme [3] does not properly implement lazy lambda calculus variable substitution and does not have the eval function, we found it very awkward to implement lazy case selection in an unextended Scheme. A simple solution to the problem of controlling evaluation order would be to implement, and use instead of Scheme, a small lambda calculus evaluator which uses the Scheme notation and which has proper lambda function semantics.

We have also found that the interactive proof editing facilities available in the Synthesizer Generator did not appear to aid substantially in the process of writing proofs. In our experience, even moderately sized proofs (proofs greater than around 100 lines) required an unreasonable amount of time to perform incremental updates on even small changes. Almost certainly part of the problem occurs as a result of the fact that the Synthesizer Generator does not provide a convenient way of dynamically changing inference scheme mechanisms. Because of this deficiency, the inference scheme machinery is written in a version of Common Lisp which introduces inefficiencies such as formula translation to Lisp s-expressions. Nevertheless, judicious use of the import and export mechanisms will allow for modular creation of large proofs by writing lemmas in one sub-proof and using them in other sub-proofs by importing them. Since such lemmas are imported as axioms, they are not checked and incremental updates are not applied to them so they do not significantly slow down the editor. Thus where modularization through lemmas is possible, the size of a sub-proof for which the system must allow convenient interactive proof updating may be much smaller than the entire proof.

Lastly, as the inference engine is implemented currently, the inference rule interpretation is immutable because the extraction machinery is written into the inference rule mechanisms. Changing interpretations for a proof might be desirable since different interpretations could be made for distinct execution machinery, while the same program-specification relationship encoding could be retained. In such cases, changing the interpretation of a proof of a specification for different execution machines corresponds roughly to a *recompilation* process: the same proof is used to create programs for multiple execution schemes. For this reason, it might be advantageous to

add the facilities to the system which allow at proof time (or perhaps link time) to specify an interpretation (from a fixed selection of interpretations, say) for the proof so that the extraction mechanisms may be changed independently from the proof. Unfortunately, adding this facility introduces numerous other problems concerning the use of extensions with such a system. Specifically, the assignment of realizations to axioms in an extension is dependent upon the interpretation of the inference rules, so the extension writer might be required to assign several different interpretations of the axioms for any particular extension. In addition, the system must contend with several interpretations of an extension's axioms, and it must ensure that the correct interpretation for the axioms is used depending upon the choice of interpretation for the inferences.

5.3 Deficiencies in the Logic

Implementation restrictions notwithstanding, MIZAR-C's logic and proof checking system lack some features which makes general program construction inconvenient and problematic. The first difficulty concerns the technical complexity involved with writing *formulas* as *type* specifications, in particular, formulas used as type specifiers can become long and their purpose obscured. In section 5.3.1 we propose an initial solution to these two problems by expanding the syntax of the variable sort specification mechanism. In section 5.3.2 we question the necessity of the special syntax for sorts when the awkwardness of using formulas to specify types can be largely alleviated through the use of tactics and *abbreviations*. An abbreviation is simply a name given to a formula. For example, we might wish to abbreviate the formula $\forall x.(\exists y.x = y)$ as the word *ident*. Then, if *ident* is known to abbreviate $\forall x.(\exists y.x = y)$, we may write *ident* whenever we wish to write $\forall x.(\exists y.x = y)$.

5.3.1 Sorts as Static Types

Restrictions on a quantifier can be interpreted as indicating parameter restrictions for partial functions specified by the quantified formula. For the sentence

$$S \equiv \forall x.Nat[x] \rightarrow \exists y.Nat[y],$$

where *Nat* is a predicate indicating membership in the naturals, we might interpret *S* to specify a function from *Nat* to *Nat*. We say that, where true, *Nat*[*x*] represents knowledge that object *x* is a member of the set of naturals, and acts as a statement of *x*'s type. The proof of the summation function in section 5.1 indicates the disadvantage associated with this kind of program-specification relationship: we must always carry around proofs of our objects' types, a process which becomes both tedious and distracting. Tactics may be defined, though, to help automate the use of these specifications in order to at least reduce the amount of tedious work associated with using formulas as types for quantifier restrictions.

```

sort := identifier
      | sort -> sort
      | < sort { , sort } >
      | (sort)

```

Figure 5.3.1: An alternative grammar for sorts.

```

for f being Nat->Nat, z being Nat holds
  (ex sum being Nat st Sum[f,z,sum])

```

Figure 5.3.2: Using special syntax for sorts.

However, it might be more convenient to build into the system a notation and interpretation for quantifier restrictions for the simple types. We may try using sorts as simple type identifiers, but since we define (see section 3.6) the sort of a complex term to be the sort of its function variable, without modification of the definitions of the sort of a term, we run into trouble. The trouble occurs because, if we interpret the sort of a term to be its type, function variables in complex terms would always have the same type as the objects in their range, a relationship which is not easily (or naturally) mapped to a realization. This problem can be solved in a straightforward way by extending the language for sort identification and changing the interpretation of sorts. In particular, we could allow sorts to be named by the grammar which appears in Figure 5.3.1. In Figure 5.3.1, *identifier* is a variable identifier and *sort* is a sort. The parenthesis as usual override the precedence of the other sort construction operators.

With the above sort syntax, we could interpret a sort composed of a simple identifier as naming a basic sort which is thought of as a built-in data type. Then we could interpret sorts of the form $s_0 \rightarrow s_1$ as naming the set of functions from sort s_0 to sort s_1 , and sorts of the form $\langle s_0, s_1, \dots, s_m \rangle, m \geq 0$ as naming the cartesian product of the m sets identified by s_i . Then a term $t \equiv (f \ t_0 \ t_1 \ \dots \ t_m), m > 0$ would be well formed if variable f is of sort $\langle s_0, s_1, \dots, s_m \rangle \rightarrow s$, and t 's sort would be s . Figure 5.3.2 gives an example of the use of this interpretation for sorts as specifying simple types. In it, $\text{Sum}[f, z, \text{sum}]$ is to be interpreted as $\text{sum} = \sum_{i=0}^z (f \ i)$, and objects satisfying the predicate Nat are said to be natural numbers. Using formulas as type specification, the formula in Figure 5.3.2 might appear as SumDef in Figure 5.3.3, which is arguably more difficult to use and state. The formula $\text{for } x \text{ st Nat}[x] \text{ holds Nat}[(f \ x)]$ acts as a certificate of the fact that f is a function from Nat to Nat since when it is true, for any object y satisfying $\text{Nat}[y]$, one can conclude that $(f \ x)$ is defined and its value also satisfies $\text{Nat}[(f \ x)]$.

It is unclear what significant advantages specialized sort syntax provides when tac-

```
for f,z st (for x st Nat[x] holds Nat[(f x)]) & Nat[z] holds
(ex sum st Nat[sum] & Sum[f,z,sum])
```

Figure 5.3.3: Using formulas to specify types.

tics combined with abbreviations may make writing and reading proofs easier without sorts. To see how the system of using formulas as types can be made more palatable without resorting to building in a typing notation, we describe two extensions to the MIZAR-C system: tactics and abbreviations.

5.3.2 Tactics and Abbreviations

The main advantage which tactics can provide is that they can automate some monotonous tasks involved in writing proofs by abstracting out proof procedures much in the same way as macros (and algorithms) abstract out computation steps. Tactics may be currently added by writing—by hand—the tactic mechanism as a named inference scheme using MIZAR-C's internal inference writing language, Lisp. For example, we have implemented a simple tactic called `munivelim` which can call `univelim` multiple times to eliminate several universal quantifiers from a formula in a single step. A tactic implemented as a named inference scheme can then be included in an extension (for use with the `extendwith` statement) which makes it convenient to bundle tactics with the axioms they work upon. Although any tactic could be implemented using Lisp, it would be convenient to allow the proof writer to add tactics to the system as they are needed, but without knowing how to write them in Lisp. In addition, it would be desirable to be able to prove each tactic is correct (i.e. that it represents a series of valid deductions using more primitive, valid inference rules) which is difficult to do if the tactics are written directly in Lisp.

The straightforward solution is to add a tactic writing syntax to the MIZAR-C language which allows for reasoning using the inference rules on named parameter formulas. Then the tactic writing mechanism could interpret the inference rule sequence used on the parameter formulas as a tactic which may be used on other actual formulas. Then, instead of extracting executable code from the proof of a tactic, we could extract a Lisp program which implements the tactic as a named inference scheme. Since these named inference schemes would be conservative extensions to the system in which they are defined, they might be added to the system where the tactic definition appears. An alternative is to export their definition using the `export` mechanism in order to load them into other proofs when an `import` is performed.

It may also be possible to encode, in an extension to the system, the inference rules as functions on MIZAR-C formulas. This extended system, call it meta-MIZAR-C, would describe the set of valid inference rules in the logic of MIZAR-C; we could then write tactics as theorems asserting the existence of new functions based on the axiomi-

tization of MIZAR-C's inference rule functions. Extractions from the (constructive) proofs of these theorems could be named and loaded into the proof checker to be used as inference schemes implementing tactics. Although this second approach, if feasible, would make simple tactic writing much more difficult, it would allow for the interesting exercise of reimplementing the base inference rules by proving them using an assumption of a definition of the logic of MIZAR-C. The potential importance of the feasibility of the latter approach is described by Constable and Howe [5] and Allen et. al. [1].

While tactics allow the writer to create succinctly written reasonings, the formulas the tactics work on can remain large and unwieldy. An abbreviation mechanism which allows formulas to be represented by a name may be a partial solution to this problem. For any large formulas which occur often, writing them as named formulas decreases the size, and increases the readability of the entire proof. Once again, using our example from Figure 5.3.3 we might wish to interpret the formula

`fFunc: for x holds Nat[x] implies Nat[(f x)];`

as a certificate that f is a function from Nat to Nat , but the meaning of the formula is not obvious and the formula itself is rather large and distracting. It would be much better to name the entire formula at `fFunc` as `Func-Nat2Nat[f]` to indicate the property of f (its type) which `fFunc` specifies. In general, we could implement abbreviations by adding a command which associates an arbitrary formula F in free variables $v_1, v_2, \dots, v_n$ ($n \geq 0$) with an unused n -place predicate $P[v_1, v_2, \dots, v_n]$ so long as $P[v_1, v_2, \dots, v_n]$ does not appear in F . Then, in the proof text which follows the abbreviation's definition, wherever predicate $P[t_1, t_2, \dots, t_n]$ appears, it would be expanded into the formula F with the n free variables v_i in F substituted with the corresponding terms t_i in $P[t_1, t_2, \dots, t_n]$. It is important to remember that this formulation of abbreviation automatically expands the abbreviation names so that the predicate $P[v_1, v_2, \dots, v_n]$ above is not actually *defined* by the abbreviation mechanism, rather it only acts to identify a formula. This distinction is important since $\mathcal{D}(F)$ may be $\emptyset$ which forces us to expand $P[v_1, v_2, \dots, v_n]$ before any inference rules such as `take` or `exintro` are used on it. As a consequence, the statement of an abbreviated formula is allowed only where the statement of the formula it abbreviates is allowed. As always, all variables in $P[t_1, t_2, \dots, t_n]$ must be defined in order for the formula it represents to be statable.

The above proposed abbreviation facility is only a partial solution for meaningfully naming large formulas since, for example, terms can become large (examine the term `(plus z (f (succ n)))` in the summation proofs in Appendix C and Appendix D). We also note that, in the natural deduction system, some inferences are written in several steps using special notation so that any one proof step, such as a universal variable declaration, may not be a complete inference. Because of this fact, some commonly occurring proof steps cannot be abstracted by a tactic because they do not constitute a complete inference. Such is the case when using a `consider` statement to eliminate an existential quantifier from a formula because the inference is completed

only when the local variable introduced by the `consider` statement is removed from consideration. For cases where common proof steps do not form a complete inference or sequence of inferences, we could instead implement *proof macros* which expand to multiple proof steps when entered into the system. Some useful proof macros might also be displayed as their macro names by using the *view* capability of the Synthesizer Generator, which allows one to display entire language strings in alternative condensed formats. Finally, the implementation of parameterized tactics, or *tacticals*, which name a family of tactics would be the next step in defining a convenient tactic mechanism. Tacticals are important for proof writing because they abstract a general inference scheme over deductions as well as formulas (see below for an example).

We now return to the question raised when discussing the sort restrictions of MIZAR-C: is the special sort syntax necessary to understand and write the our desired proofs with reasonable comfort? As perhaps an indicator of the comfort which tactics and abbreviations might add to the system, we show the example proof which appears in Appendix C rewritten in Appendix D using the tacticals `elim` and `capture` and the (currently imaginary) abbreviations `SigmaF` and `NatFunc`. The proof uses the abbreviations

$$\begin{aligned}\text{SigmaF}[z,n,f] &\hat{=} \text{Nat}[z] \ \& \ \text{Sigma}[z,n,f] \\ \text{NatFunc}[f] &\hat{=} \text{for } x \text{ st } \text{Nat}[x] \text{ holds } \text{Nat}[(f \ x)]\end{aligned}$$

where $\hat{=}$ means the name on the left hand side abbreviates the formula on the right hand side. Both tacticals may be implemented in Lisp (`elim` already has been) and added to an extension and perform the expected inferences: `capture` introduces an existential formula of the form `ex v st v=t` for some argument term `t`, and `elim` performs multiple universal quantifier elimination and implication elimination given a sequence of argument variables, a universal sentence, and formulas matching antecedents for implications in the universal sentence. Their definitions are clarified by their use in the example proof.

Bibliography

- [1] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–107, Los Alamitos, CA, 1990. IEEE Computer Society Press.
- [2] Michael J. Beeson. *Foundations of Constructive Mathematics*, volume 6 of *Ergebnisse der Mathematik und ihrer Grenzgebiete 3.Folge*. Springer-Verlag, Berlin-Heidelberg-New York, 1985.
- [3] William Clinger and Jonathan Rees (eds.). Revised⁴ report on the algorithmic language scheme. *LISP Pointers*, 4(3):1–56, 1991.
- [4] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [5] Robert L. Constable and Douglas J. Howe. Nuprl as a general logic. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 77–90. Academic Press, London, UK, 1990.
- [6] Thierry Coquand. On the analogy between propositions and types. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, pages 399–418, Reading, Massachusetts, 1990. Addison-Wesley.
- [7] N. G. de Bruijn. A survey of the project AUTOMATH. In *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606, London, UK, 1980. Academic Press.
- [8] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1992.
- [9] Timothy G. Griffin. A formal account of notational definition. Technical report, Cornell University, 1988.

- [10] Susumu Hayashi. An introduction to PX. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, pages 431–486, Reading, Massachusetts, 1990. Addison-Wesley.
- [11] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. The MIT Press, Cambridge, Massachusetts, 1989.
- [12] Martin C. Henson. Realizability models for program construction. In G. Goos and J. Hartmanis, editors, *Mathematics of Program Construction, Proceedings*, volume 375 of *Lecture Notes in Computer Science*, pages 256–72, Berlin-Heidelberg-New York, June 1989. Springer-Verlag.
- [13] Martin C. Henson and Raymond Turner. A constructive set theory for program development. In G. Goos and J. Hartmanis, editors, *Foundations of Software Technology and Theory of Computer Science, Proceedings*, volume 338 of *Lecture Notes in Computer Science*, pages 329–47, Berlin-Heidelberg-New York, December 1988. Springer-Verlag.
- [14] H. James Hoover and Piotr Rudnicki. *Lecture Notes on Formal Systems and Logic in Computing Science*. Department of Computing Science, University of Alberta, 1991. Course Notes.
- [15] Paul Francis Mendler. Inductive definition in type theory. Technical report, Cornell University, 1987.
- [16] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type theory*, volume 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford, 1990.
- [17] Larry Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1987.
- [18] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.
- [19] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin-Heidelberg-New York, 1989.
- [20] Grant Malcolm Roland Backhouse, Paul Chisholm and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.
- [21] Piotr Rudnicki and Włodzimierz Drabent. Proving properties of pascal programs in MIZAR 2. *Acta Informatica*, 22:311–331, 1985.

- [22] Yukihide Takayama. QPC: QJ-based proof compiler – simple examples and analysis. In G. Goos and J. Hartmanis, editors, *Second European Symposium on Programming, Proceedings*, Lecture Notes in Computer Science, pages 49–63, Berlin-Heidelberg-New York, 1988. Springer-Verlag.
- [23] Yukihide Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.
- [24] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction, vols I and II*, volume 121 & 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1988.

Appendix A

Annotated proof of Pred

```
article pred
environ
  extendwith naturals-t;  /* Extend with definition of naturals */

begin
  /* Restated axioms from naturals-t */
  AX-successor:
    for x st Nat[x] holds Nat[(succ x)] by direct(NatT-succ);
    /* Is realized by (LAMBDA (x) (LAMBDA (nat) #t)) since it is */
    /* a higher order function on a natural witness transformation */

  AX-zero-nat:
    Nat[0] by direct(NatT-0);
    /* Nat[0] is realized by #t */

  0=0 by eqintro();
  /* 0 is defined in naturals-t; 0=0 is realized by #t */

  (succ 0)=0 or 0=0 by disjintro(_PREVIOUS);
  /* 0=0 is remembered as the (only) true disjunct, realized by a */
  /* list of the disjunct realizations: (LIST #f #t) */

  Nat[0] & ((succ 0)=0 or 0=0) by conj(_PREVIOUS, AX-zero-nat);
  /* The new conjunction realization remembers the realizations of */
  /* both conjuncts in a list: (LIST #t (LIST #f #t)) */

  BaseCase:
    ex pred st Nat[pred] & ((succ pred)=0 or 0=0) by exintro(_PREVIOUS);
    /* We save the value 0 as the one satisfying the base case, and */
    /* the witness that it satisfies it: */
    /* BASECASE = (LIST 0 (LIST #t (LIST #f #t))) */
```

```

/* Our goal in the induction step is to prove
 *   GOAL: for x st Nat[x] holds
 *       (ex pred st Nat[pred] & ((succ pred)=x or x=0)) implies
 *       (ex pred st Nat[pred] & ((succ pred)=(succ x) or (succ x)=0))
 */
InductionStep:
  now
    let x;
      /* Parameter named x */
    assume xNat: Nat[x];
      /* Precondition Parameter nat-x */
    assume unusedIH: ex pred st Nat[pred] & ((succ pred)=x or x=0);
      /* Precondition Parameter ind-hyp, the induction hypothesis */

    /* We can construct our desired conclusion without reference to */
    /* the induction hypothesis using the axioms for naturals      */

    Nat[x] implies Nat[(succ x)]          by univelim(AX-successor);
      /* Eliminate on x,      (AX-successor x) == (LAMBDA (nat) #t) */

    Nat[(succ x)]                        by impelim(_PREVIOUS, xNat);
      /* Eliminate on Nat[x], ((LAMBDA (x)) #t) == #t          */

    (succ x) = (succ x)                  by take(_PREVIOUS);
      /* (succ x) is a valid term, equality realized by #t      */

    (succ x) = (succ x) or (succ x)=0    by disjintro(_PREVIOUS);
      /* Remember the realization of true disjunct (LIST #t #f) */

    Nat[x] & ((succ x)=(succ x) or (succ x)=0) by conj(_PREVIOUS, xNat);
      /* Remember conjuncts' realizations: (LIST nat-x (LIST #t #f)) */

    /* The "thus" statement identifies the formula which is to be */
    /* concluded as a result of this reasoning */

    thus ex pred st Nat[pred] & ((succ pred) = (succ x) or (succ x)=0)
      by exintro(_PREVIOUS);
      /* Value remembered for pred is x, the parameter:          */
      /* (LIST x (LIST nat-x (LIST #t #f)))                      */
    end;

  /* The assumptions and local variable declarations result in
   * abstraction through implication introduction and universal
   * introduction respectively when we move out of the "now--end".

```

```

* We have proven GOAL:, from the proof, it is realized by:
*   INDSTEP =
*     (LAMBDA (x)
*       (LAMBDA (nat-x)
*         (LAMBDA (ind-hyp)
*           (LIST x (LIST nat-x (LIST #t #f))) )))
*/
PredFunc:
  for x st Nat[x] holds
    (ex pred st Nat[pred] & ((succ pred)=x or x=0))
  by natind(BaseCase, InductionStep);
/* The realizer for the formula concluded by the induction scheme natind
* is a function which calls a recursor, NAT_IND_ITER to apply
* INDSTEP iteratively to BASECASE:
* (LAMBDA (x)
*   (LAMBDA (nat-x)
*     (NAT_IND_ITER x BASECASE INDSTEP)))
*
* in more familiar Scheme, we can rewrite the realizer as the recursive
* program:
*
* (define (rec x nat-x)
*   (if (= x 0)
*       BASECASE
*       ( (INDSTEP (rec (- x 1) nat-x)) nat-x)
*   ))
*/

```

Appendix B

Non-constructive Inference Rules

These are the inherently non-constructive inference rules which implement standard classical inference rules. In the following A and B are arbitrary formulas.

$$\text{negelim} \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}$$

$$\text{contrintro} \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \perp}$$

$$\text{negintro} \frac{\Gamma \vdash A \rightarrow \perp}{\Gamma \vdash \neg A}$$

$$\text{revimpl} \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash \neg B}{\Gamma \vdash \neg A}$$

$$\text{exmiddle} \frac{}{\vdash \vee \langle A, \neg A \rangle}$$

Let C be a formula such that either $C = \neg A$ or $\neg C = A$, then

$$\text{imp2disj} \frac{\Gamma \vdash A \rightarrow B}{\Gamma \vdash \vee \langle B, C \rangle}$$

$$\text{disj2imp} \frac{\Gamma \vdash \vee \langle B, C \rangle}{\Gamma \vdash A \rightarrow B}$$

Appendix C

Proof of Summation

```
article summation
environ

extendwith naturals-t;
extendwith nat-plus-t;

begin

{ A 'definition' of the summation operator as a predicate. }
SigmaDef:
  for f st (for x st Nat[x] holds Nat[(f x)]) holds
    Sigma[(f 0), 0, f]
    & (for n, z st Nat[n] & Nat[z] holds
      Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f])
  by magic();

{ Restatement of axioms in naturals-t and nat-plus-t for reference }

RNatT-0:      Nat[0]
  by direct(NatT-0);
RNatT-succ:   for x st Nat[x] holds Nat[(succ x)]
  by direct(NatT-succ);
RNatT-plus:   for x,y st Nat[x] & Nat[y] holds Nat[(plus x y)]
  by direct(NatT-plus);

now
  let f; assume fFunc: for x st Nat[x] holds Nat[(f x)];

  (for x st Nat[x] holds Nat[(f x)]) implies
    Sigma[(f 0), 0, f]
    & (for n, z st Nat[n] & Nat[z] holds
```

```

    Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f]
  by univelim(SigmaDef);

1: Sigma[(f 0), 0, f] &
  (for n, z st Nat[n] & Nat[z] holds
    Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f])
  by impelim(_PREVIOUS, fFunc);

2: Sigma[(f 0), 0, f] by conj(1);

3: for n, z st Nat[n] & Nat[z] holds
    Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f]
  by conj(1);

{ Show a value exists for the base case for 0 }
BaseCase: now
  Nat[0] implies Nat[(f 0)]                by univelim(fFunc);
  nf0: Nat[(f 0)]                          by impelim(_PREVIOUS, RNatT-0);
  Nat[(f 0)] & Sigma[(f 0), 0, f]           by conj(nf0, 2);
  thus ex z st Nat[z] & Sigma[z, 0, f]     by exintro(_PREVIOUS);
end;

{ Now induction step }
IndStep: now
  let n;
  assume nNat: Nat[n];
  assume IndHyp: ex z st Nat[z] & Sigma[z, n, f];

  consider z such that 4: Nat[z] & Sigma[z, n, f] by direct(IndHyp);
  nats: Nat[n] & Nat[z] by conj(nNat, 4);
  sig: Sigma[z, n, f] by conj(4);

  for z st Nat[n] & Nat[z] holds
    Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f]
  by univelim(3);

  Nat[n] & Nat[z] implies
    (Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f])
  by univelim(_PREVIOUS);

  Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f]
  by impelim(_PREVIOUS, nats);

sigp: Sigma[(plus z (f (succ n))), (succ n), f]
  by impelim(_PREVIOUS, sig);

```

```

(f (succ n)) = (f (succ n))      by take(_PREVIOUS);
consider fsucc such that
  vfsucc: fsucc = (f (succ n))    by exintro(_PREVIOUS);
  (succ n) = (succ n)              by take(_PREVIOUS);
consider succn such that
  vsuccn: succn = (succ n)         by exintro(_PREVIOUS);
  Nat[n] implies Nat[(succ n)]    by univelim(RNatT-succ);
  Nat[(succ n)]                   by impelim(_PREVIOUS,nNat);
nsn: Nat[succn]                   by equality(_PREVIOUS,vsuccn);
  Nat[succn] implies Nat[(f succn)] by univelim(fFunc);
  Nat[(f succn)]                  by impelim(_PREVIOUS,nsn);
  Nat[(f (succ n))]               by equality(_PREVIOUS,vsuccn);
  Nat[fsucc]                      by equality(_PREVIOUS,vfsucc);
nfsz: Nat[z] & Nat[fsucc]          by conj(_PREVIOUS,4);
  for fsucc st Nat[z] & Nat[fsucc] holds
    Nat[(plus z fsucc)]            by univelim(RNatT-plus);
    Nat[z] & Nat[fsucc] implies
      Nat[(plus z fsucc)]          by univelim(_PREVIOUS);
      Nat[(plus z fsucc)]          by impelim(_PREVIOUS,nfsz);
      Nat[(plus z (f (succ n)))]   by equality(_PREVIOUS,vfsucc);
      Nat[(plus z (f (succ n)))] &
      Sigma[(plus z (f (succ n))), (succ n), f]
      by conj(_PREVIOUS,sig);
  thus ex z st Nat[z] & Sigma[z, (succ n), f] by exintro(_PREVIOUS);
end;

thus
  for n st Nat[n] holds (ex z st Nat[z] & Sigma[z, n, f])
    by natind(BaseCase, IndStep);
end;

SigmaFunc:
for f st (for x st Nat[x] holds Nat[(f x)]) holds
  (for n st Nat[n] holds (ex z st Nat[z] & Sigma[z, n, f]))
  by direct(_PREVIOUS);

```

Appendix D

Proof of Summation using Tactics and Abbreviations

```
now
  let f; assume fFunc: NatFunc[f];

  1: Sigma[(f 0), 0, f] &
    (for n, z st Nat[n] & Nat[z] holds
      Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f])
    by elim[f](SigmaDef, fFunc);

  2: Sigma[(f 0), 0, f] by conj(1);
  3: for n, z st Nat[n] & Nat[z] holds
      Sigma[z, n, f] implies Sigma[(plus z (f (succ n))), (succ n), f]
    by conj(1);

{ Show a value exists for the base case for 0 }
BaseCase: now
  nf0: Nat[(f 0)]                                     by elim[0](fFunc, RNatT-0);
      Nat[(f 0)] & Sigma[(f 0), 0, f]                 by conj(nf0, 2);
  thus ez z st SigmaF[z, 0, f]                         by exintro(_PREVIOUS);
end;

{ Now induction step }
IndStep: now
  let n; assume nNat: Nat[n];
  assume IndHyp: ez z st SigmaF[z, n, f];

  consider z such that 4: SigmaF[z, n, f] by direct(IndHyp);
  nats: Nat[n] & Nat[z] by conj(nNat, 4);
  sig: Sigma[z, n, f] by conj(4);
```

```

sigp: Sigma[(plus z (f (succ n))), (succ n), f] by elim[n,z](3,nats,sig);
  consider fsucc such that
    vfsucc: fsucc = (f (succ n)) by capture[(f (succ n))](PREVIOUS);
  consider succn such that
    vsuccn: succn = (succ n)      by capture[(succ n)](PREVIOUS);
    Nat[(succ n)]                 by elim[n](RNatT-succ,nNat);
nnsn: Nat[succn]                  by equality(PREVIOUS,vsuccn);
    Nat[(f succn)]                by elim[succn](fFunc,nnsn);
    Nat[(f (succ n))]             by equality(PREVIOUS,vsuccn);
    Nat[fsucc]                    by equality(PREVIOUS,vfsucc);
nfsz: Nat[z] & Nat[fsucc]         by conj(PREVIOUS,4);
    Nat[(plus z fsucc)]           by elim[z,fsucc](RNatT-plus,nfsz);
    Nat[(plus z (f (succ n)))]    by equality(PREVIOUS,vfsucc);
    SigmaF[(plus z (f (succ n))), (succ n), f]
                                by conj(PREVIOUS,sigp);
  thus ez z st SigmaF[z,(succ n),f] by exintro(PREVIOUS);
end;

thus
  for n st Nat[n] holds (ex z st SigmaF[z,n,f])
    by natind(BaseCase, IndStep);
end;

SigmaFunc:
  for f st NatFunc[f] holds
    (for n st Nat[n] holds (ex z st SigmaF[z,n,f]))
    by direct(PREVIOUS);

```